

Together We Go Further: LLMs and IDE Static Analysis for *Extract Method* Refactoring

DORIN POMIAN, University of Colorado Boulder, USA
 ABHIRAM BELLUR, University of Colorado Boulder, USA
 MALINDA DILHARA, University of Colorado Boulder, USA
 ZARINA KURBATOVA, JetBrains Research, Serbia
 EGOR BOGOMOLOV, JetBrains Research, the Netherlands
 TIMOFEY BRYKSIN, JetBrains Research, Cyprus
 DANNY DIG, JetBrains Research, University of Colorado Boulder, USA

Excessively long methods that encapsulate multiple responsibilities within a single method are challenging to comprehend, debug, reuse, and maintain. The solution to this problem, a hallmark refactoring called *Extract Method*, 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. Choosing which statements are profitable to extract has been the subject of many research tools that employ hard-coded rules to optimize software quality metrics. 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, they are more likely to imitate human behavior. If we harness the familiarity of LLMs 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 augmenting the refactoring capabilities of IDEs with the power of LLMs to perform *Extract Method*. Our formative study on 2,849 *Extract Method* scenarios revealed that LLMs are very effective in giving expert suggestions, yet they are unreliable: up to 62.3% of the suggestions are hallucinations. We propose a novel approach that synergistically combines the creative potential of LLMs with static analysis to enhance the refactoring suggestions generated by LLMs. Additionally, we utilize the safety measures of static analysis within IDEs to execute refactorings safely. Starting from candidates suggested by LLMs, we filter, further enhance, and then rank suggestions based on static analysis techniques from program slicing. We designed, implemented, and evaluated this approach in an IntelliJ IDEA plugin called *EM-Assist*. We empirically evaluated *EM-Assist* on a diverse, publicly available corpus that other researchers used in the past. The results show that *EM-Assist* outperforms previous state-of-the-art tools: at 1% tolerance, *EM-Assist* suggests the correct refactoring among its top-5 suggestions 60.6% of times, compared to 54.2% reported by existing ML models, and 52.2% reported by existing static analysis tools. When we replicated 2,849 actual *Extract Method* instances from open-source projects, *EM-Assist*'s recall rate was 42.1% compared to 6.5% for its peers. Furthermore, we conducted firehouse surveys with 20 industrial developers and suggested refactorings on their recent commits. 81.3% of the respondents agreed with the recommendations provided by *EM-Assist*. This shows the usefulness of our approach and ushers us into a new era of refactoring when LLMs become effective AI assistants for developers.

1 INTRODUCTION

Excessively long methods that encapsulate multiple responsibilities within a single method are challenging to comprehend, debug, reuse, and maintain [10, 27, 61]. To mitigate these issues, software developers use *Extract Method* refactoring – a hallmark refactoring [27] supported by all modern IDEs. This refactoring involves moving a block of statements from a host method to a

Authors' addresses: Dorin Pomian, University of Colorado Boulder, Boulder, Colorado, USA, dorin.pomian@colorado.edu; Abhiram Bellur, University of Colorado Boulder, Boulder, Colorado, USA, abhiram.bellur@colorado.edu; Malinda Dilhara, University of Colorado Boulder, Boulder, Colorado, USA, malinda.malwala@colorado.edu; Zarina Kurbatova, JetBrains Research, Serbia, zarina.kurbatova@jetbrains.com; Egor Bogomolov, JetBrains Research, the Netherlands, egor.bogomolov@jetbrains.com; Timofey Bryksin, JetBrains Research, Cyprus, timofey.bryksin@jetbrains.com; Danny Dig, JetBrains Research, University of Colorado Boulder, USA, danny.dig@colorado.edu.

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 as being among the top-5 most frequently performed in practice [43, 45, 62] both for manual and automated refactoring.

The process of performing an *Extract Method* 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. The research community developed various techniques for suggesting these. Some research tools employ static analysis [14, 39, 60, 61, 67], while others employ machine learning (ML) models [18, 66]. These tools employ various software quality metrics (e.g., cohesion of statements), which they maximize either based on heuristics or by training ML classifiers.

While these tools excel in adhering to software engineering principles like the Single Responsibility Principle [38], they exhibit relatively low recall rates when evaluated on real-world *Extract Method* instances. We posit that the decision to refactor is both *a science and an art*. When refactoring, 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 can be a reason for the developers’ reluctance to use automated refactoring support [57] and can explain the large gap between high metrics scores of refactoring recommendation approaches and their low acceptance by developers [20–22, 52].

Recently, Large Language Models (LLMs) [24, 29, 46] are emerging as powerful companions for software engineering tasks: code completion [15], code summarization [24], commit message generation [2], code transformation [19, 49], reproducing bug reports [54], etc. 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 are likely to suggest refactorings that developers would accept. Our formative study on 2,849 *Extract Method* scenarios revealed that LLMs are very effective in giving expert suggestions. They are also very prolific, producing 12,387 suggestions for 2,849 *Extract Method* scenarios. However, we also discovered that LLMs are unreliable. 62.3% of their suggestions are *hallucinations*, i.e., they seem plausible at first, but are actually deeply flawed: 45.7% of the suggested refactorings contain code fragments that are invalid to extract (e.g., would produce compile errors), and 16.6% are not useful (e.g., the suggestion includes the whole body of the host method).

To advance the state of the art and practice for refactoring, we aim to 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. Thus, we have designed, implemented, and evaluated *EM-Assist*, an IntelliJ IDEA plugin, that supports Java and Kotlin *Extract Method* 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 employs three ranking mechanisms to prioritize 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. First, we determine the effectiveness of several LLMs in generating refactoring recommendations. Our results show that LLMs are prolific, generating on average 20 suggestions per method, yet on average, 11 are hallucinations. Second, we conduct a *sensitivity analysis* to tune hyper-parameters of the LLMs (e.g., the “temperature” which controls the degree of generations’ variability) and to measure the improvements brought by our enhancements. Third, to determine the effectiveness of *EM-Assist*, we use a diverse, publicly available corpus [66] that other researchers used in the past. Furthermore, we replicated 2,849 of actual *Extract Method* instances from open-source projects. The results show that *EM-Assist* outperforms state-of-the-art static-analysis tools such as JDeodorant [61], JExtract [56], SEMI [14], LiveRef [25, 26], and also outperforms machine learning-based techniques like REMS [18] and GEMS [66]. *EM-Assist* is able to suggest the ground truth extract method 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.

Moreover, to assess the practicality and effectiveness of refactoring recommendations generated by *EM-Assist*, we employed firehouse surveys [44] with 16 industrial developers from a reputable software company. We contacted 20 developers and presented them with refactoring suggestions for lengthy methods they previously committed into their software repository. 80% of the developers responded, with 81.3% of respondents agreeing 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.

In summary, this paper makes the following key contributions:

- (1) To the best of our knowledge, we present the first approach to synergistically combine the creativity of LLMs with the safety of static analysis and IDEs. Starting from candidates suggested by LLMs, we filter, further enhance, and then rank suggestions based on static analysis. Thus, we effectively bridge the gap between refactoring recommendation and application, and the gap between refactoring suggestions and developer practices.
- (2) We discovered best practices for prompting LLMs to produce effective refactoring suggestions and we reveal LLMs’ strengths and weaknesses for refactoring tasks.
- (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 2,849 real-world refactorings shows that *EM-Assist* outperforms traditional static analysis-based techniques and machine learning approaches in automatically suggesting and performing *Extract Method* refactoring. Moreover, our survey with 16 industrial developers provides insights about the reasons why Java developers accept or reject suggestions generated by *EM-Assist*.

To aid reproducibility in SE research, we make *EM-Assist*, a demo, and all our experimental data available anonymously [7].

2 MOTIVATING EXAMPLE

We illustrate the challenges of suggesting code fragments to extract into new methods in a way that humans find useful, as well as the process of unleashing the full potential of LLMs for *Extract Method* refactoring using a real-world example. Figure 1 shows an *Extract Method* refactoring performed by Neo4J developers. They extracted statements in lines 157 and 158 into a new method

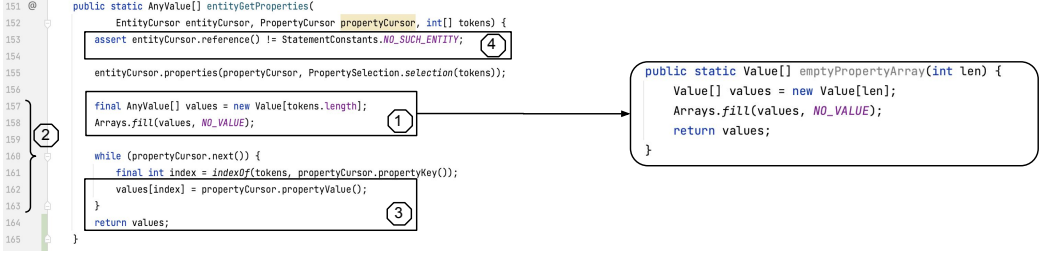


Fig. 1. The numbered code snippets represent 1: an extract function refactoring in the project Neo4j, commit a05a8c5, 2: a suggestion produced by static analysis tool, 3: a hallucination suggested by an LLM, 4: a not useful suggestion produced by LLM.

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 [14, 39, 61] for suggesting *Extract Method* 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 [22, 52].

Pre-trained Large Language Models (LLMs) signify a pivotal shift in the landscape of natural language processing. These models undergo rigorous training on extensive and diverse datasets, including terabytes of texts. In addition to books, articles, and texts from the web, modern datasets also comprise source code at GitHub scale [34, 37]. Training on a broad range of texts leads to models that are capable in a wide range of language and code processing tasks [12, 46], all orchestrated through prompt engineering [50, 64]. Prompt engineering involves the skillful formulation of natural language prompts to direct LLMs’ behavior and responses to solve the task at hand.

Moreover, LLMs stand out as formidable instruments, particularly in the domains of source code generation and refinement. Their ability is evident in their capacity to suggest code improvements, including the detection and rectification of “code smells” [9, 27]. Since LLMs are trained on vast collections of real-world code, they can capture common patterns that humans use when writing code. We hypothesize that LLMs can effectively suggest which statements should form new methods based on the patterns of which statements commonly go together. This can narrow the gap between automated suggestions and human intuition.

We revisit the refactoring scenario, only this time we involve an LLM, specifically opting for GPT-3.5 [12] as our choice, developed by OpenAI. We provided the LLM with a prompt that included the code snippet in the host method `entityGetProperties`, followed by a description of the *Extract Method* 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. In this case, 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

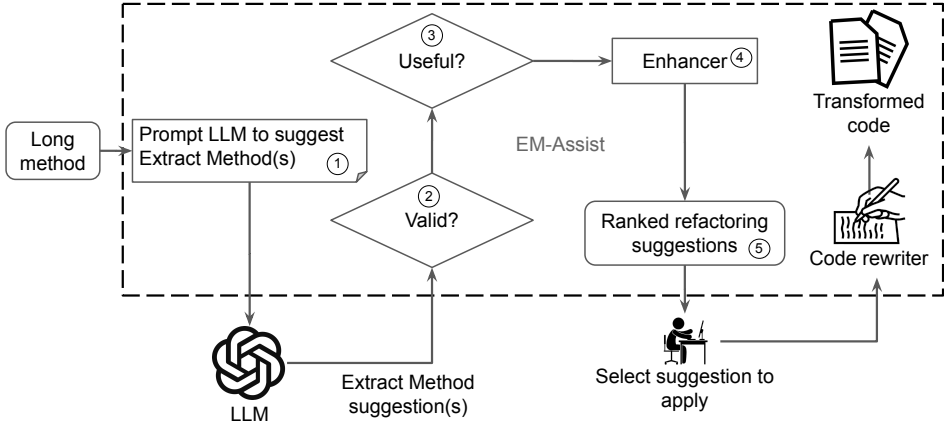


Fig. 2. The workflow of generating refactoring suggestions and then applying them with *EM-Assist*.

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*. For our scenario, the LLM produced 3 invalid suggestions. While the LLM might have seen those code statements being collocated and associated in real-world codebases, because the LLM does not fully understand the semantics of *Extract Method* and syntactical rules of well-formed code, those associations would 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 4.3, 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 experiment with various LLMs (in Section 4.2 we tried three different LLMs), altogether resulting in a combinatorial explosion of experiments.

This is where our tool, *EM-Assist*, comes alongside to liberate the developers so they can focus on the creative part: using their expertise to decide which of a small number of high-quality suggestions to accept. Next, we present the workflow of using *EM-Assist* and its core architecture.

3 TECHNIQUE

In this section, we present the workflow that our novel approach and tool, *EM-Assist*, uses to automatically suggest and perform *Extract Method* refactoring for long methods in Java and Kotlin codebases. Figure 2 shows the architecture and the steps performed by *EM-Assist*. First, *EM-Assist* invokes the LLM by prompting it to generate *Extract Method* suggestions for the selected long methods (① in Figure 2). We further elaborate on this in Section 3.1. Following this, our algorithm filters out hallucinations and retains *applicable* refactoring options that are free of errors (② in Figure 2) and are useful (③ in Figure 2). We discuss these in detail in Section 3.2 and Section 3.3. These steps play a key role in selecting realistic 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) the quality of the suggestions by expanding or shrinking the code fragment in the suggestion based on program slicing rules presented in Section 3.4. These rules ensure that refactorings better align with the developer’s intentions. As there could be several alternatives (which can overwhelm the user) for how to split a long method, *EM-Assist* ranks the candidates (⑤ in Figure 2) (see Section 3.5), before it presents a high-quality subset to the user. The 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. You can find *EM-Assist*, instructions for its installment, and example of its working in the supplementary material [7].

Definition 3.1. (Long method) denoted as $l_i(l_s, l_e)$, in a codebase is quantified as $L(l_i)$, where $L(l_i)$ represents the length of the method. This length is determined as the difference between the starting line number of the method in the file (l_s) and the ending line number of the method (l_e), accounting for spaces and comments while excluding Javadoc comments. A method l_i is classified as “long” if it exceeds a predefined threshold, such that $L(l_i) > TV$, with TV signifying the specified threshold value. We determine this threshold value empirically, and we use it to define the minimum length at which a method is considered “long” within the codebase.

Definition 3.2. (Extract Method) denoted as $e_i(n_i, (s_i, f_i))$, where n_i represents the method name, (s_i, f_i) signifies a code fragment within a long method l_i . It is characterized by the starting line number, defined as $s_i > l_s$, and the ending line number, defined as $e_i < l_e$. These line numbers are calculated with respect to the original source code file in which the host method l_i is located.

Definition 3.3. (Extract Method Suggestions (\mathcal{S})) is a set of extract method suggestions generated by an LLM for a long method l_m , formally, $\mathcal{S} = \{e_i(n_i, (s_i, f_i)), e_j(n_j, (s_j, f_j)), \dots\}$

Definition 3.4. (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 3.5. (Not Useful Extract Methods (\mathcal{NU})) is a subset of \mathcal{V} that does not satisfy the usefulness conditions ($U(e_i)$), ensuring that the suggestions are neither too large, encompassing almost the entire method body, nor too small, essentially one-liners. Formally, $\mathcal{NU}_i = \{e_i \in \mathcal{V} \mid \neg U(e_i)\}$.

Definition 3.6. (Applicable Extract Methods (\mathcal{A})) constitute a set of *Extract Method* suggestions within a long method l_i that have been filtered to exclude both invalid and not useful suggestions. Formally, $\mathcal{A}_i = \{e_i \in \mathcal{S} \mid e_i \notin (\mathcal{I}_i \cup \mathcal{NU}_i)\}$.

3.1 Generating *Extract Method* Suggestions

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

3.1.1 Prompt engineering. LLMs are versatile models capable of various applications, but they require specific preparation when used for a particular task [54]. To facilitate this, we employ *in-context learning* [64], 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 [12, 48] by incorporating a set of 2 examples into the prompt, following best practices as discussed by [28]. The prompt includes: (i) A succinct overview of *Extract Method* 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 [7].

3.1.2 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) [12, 36, 48], 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 0 produce focused and less diverse results. We determine the optimal value empirically (see Section 4.3).

Even when identical prompts are presented multiple times, the output can exhibit variations due to a couple of factors: (i) LLMs, such as GPT-3.5 and GPT-4, employ a combination of learned patterns and random sampling in generating responses, introducing slight deviations in their answers each time due to inherent randomness; and (ii) LLMs possess the capability to explore various potential solutions and refine their responses iteratively through interactions and feedback [12, 48], continually enhancing their outputs in subsequent iterations. Therefore, the quality and quantity of predictions made by an LLM are influenced by the temperature setting and the frequency of prompting the same prompt. In Section 4.3, we conduct a *sensitivity analysis* to determine how the performance of *EM-Assist* varies with these settings. This analysis enables *EM-Assist* to utilize the optimal settings for generating refactoring suggestions.

3.2 Removing Invalid *Extract Method* Suggestions

Not all the suggestions generated by LLMs can be directly applied to codebases. Some suggestions may result in non-compilable code (e.g., ③ in our motivating example from Figure 1), which are hallucinations that developers are unlikely to accept [47].

Scope Analysis: As the first check, *EM-Assist* analyzes the suggestion to ensure that it can be extracted without breaking the code’s scope. The code should be self-contained, with all necessary variables and objects accessible within the same scope or passed as parameters. 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.

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

(i) **Return Values:** If the code fragment to be 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.

(ii) *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.

(iii) *Control Flow*: *EM-Assist* reasons about the control flow within the code fragment in the 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 Platform¹ (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 *Extract Method* 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 (45.7% of all suggestions in our formative study).

3.3 Removing Refactoring Suggestions That Are Not Useful

Once we retain only valid refactoring suggestions, the next step is to filter out those suggestions that are not useful and retain 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 *Extract Method* suggestions that encompass 86% or more of the statements present in the long method. (ii) Following practices adopted by other researchers [25, 30, 61, 68], we exclude *Extract Method* suggestions that are one-line code extractions.

Extracting an entire method or a one-liner into another method is, in general, not useful for the purpose of improving the existing code. This might be useful if the developers plan to introduce new code statements and implement new features into the host method or the extracted method that expand the original functionality. Notice that we designed *EM-Assist* for *code renovation* and not for the feature expansion use case.

3.4 Enhancing Refactoring Suggestions

After identifying applicable suggestions devoid of hallucinations, the subsequent step involves refining and enhancing these selections. During this phase, we apply the following enhancements to further improve the quality of the valid suggestions.

(i) *Program slicing*: Taking inspiration from tools that rely solely on static analysis, such as those utilizing program slicing [1, 14, 39, 40, 61], 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 represent the first statement of the code fragment to be extracted into a new method. Let EF_{s-1} represent a variable declaration statement that initializes a variable v that is used in the code fragment of the suggested extract method. v is called a live variable, determined through a def-use chain. In such cases, we include 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 to the extracted method. In the future, we plan to experiment with other such heuristics based on the feedback we receive from users of *EM-Assist*.

(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

¹The IntelliJ Platform: <https://www.jetbrains.com/opensource/idea/>.

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 4.3.

3.5 Ranking Refactoring Suggestions

The number of the remaining applicable suggestions per host method can be large (on average 7 per method as shown in RQ1) and overwhelm the developer. To prioritize high-quality suggestions, we implemented a ranking mechanism. We use three ranking functions, denoted as $\mathcal{T}_j(e_i)$, where j ranging from 1 to 3. Each function assigns a score to each *Extract Method* suggestion, resulting in three distinct rankings for each suggestion based on these scores. We present the three ranking functions below:

$$\mathcal{T}_1(e_i) = \mathcal{H}(e_i) = \sum_{i=1}^N \mathcal{F}(\text{line}_i) \quad \mathcal{T}_2(e_i) = \mathcal{P}(e_i) \quad \mathcal{T}_3(e_i) = \mathcal{H}(e_i) \cdot \mathcal{P}(e_i)$$

Where:

N : Total number of lines in the *Extract Method* suggestion.

$\mathcal{F}(\text{line}_i)$: Number of times line_i appears among all the other suggestions made by the LLM.

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

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

In Ranking Function 1 ($\mathcal{T}_1(e_i)$), we assign scores to suggestions based on the “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. We calculate a “heat” score for each applicable refactoring suggestion. To compute the heat map for a specific host method, we follow a systematic process. First, we iterate through each line of the host method, recording 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. Subsequently, we aggregate these individual line scores to derive the overall heat score for each suggestion (which comprises several lines of code). Then, we rank suggestions according to their heat score.

Ranking Function 2 ($\mathcal{T}_2(e_i)$) ranks suggestions by taking into account their popularity. Remember that we re-prompt the LLM several times with the same prompt (as discussed in Section 3.1.2). 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 a substantial volume of source code, enabling them to mimic how real developers construct methods, we give precedence to suggestions that appear repeatedly through this ranking function.

Ranking Function 3 ($\mathcal{T}_3(e_i)$) 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 4.3.

4 EVALUATION

In this section, we empirically evaluate *EM-Assist* by answering the following research questions:

RQ1. How effective are LLMs at generating refactoring suggestions? Answering this is important because this is the first paper that uses LLMs for a novel usage scenario. We conducted a quantitative analysis to measure the effectiveness of LLMs in generating *Extract Method* refactoring suggestions using three of the latest and most extensive LLMs available.

RQ2. How do refactoring suggestions change with different LLM parameters? This is important because we are discovering best practices that others can use for unleashing the full

potential of LLMs for refactoring tasks. Moreover, studying how an LLM adjusts refactoring suggestions to different LLM parameters is crucial for tool integration. To answer this question, we conduct a *sensitivity analysis* to assess the quality of refactoring suggestions in terms of the recall rate among top-5 suggestions (Recall@5) across various LLM parameters and the contributions brought by our enhancements and rankings.

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 *Extract Method* refactorings. We compare these tools in terms of Recall@5 using the same corpus that they previously used.

RQ4. How useful are the provided recommendations to developers? As we want to advance the science and practice of refactoring, and because *EM-Assist* is a code renovation tool that developers use interactively, 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. Then, we report their responses regarding the refactoring recommendations proposed by *EM-Assist*.

4.1 Datasets

To answer our research questions, we use two kinds of 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. Within the Community corpora, we distinguish between two groups: Community Corpus-A and Community Corpus-B, mutually exclusive, each built and maintained by distinct groups of researchers.

(i) *Community Corpus-A*: consists of 122 Java methods and their corresponding *Extract Method* refactorings collected from five open-source repositories: MyWebMart, SelfPlanner, WikiDev, JHotDraw, and JUnit. This dataset previously served as the foundation for evaluating various state-of-the-art *Extract Method* refactoring automation tools, including *JExtract* [56], *JDeodorant* [61], *SEMI* [14], *GEMS* [66], and *REMS* [18]. While Community Corpus-A is the most widely used by other researchers, it might be interpreted as being subjective. It contains a mix of (i) hypothetical refactoring scenarios in which human experts determined which statements should be extracted from host methods in open source projects and (ii) *synthetic* refactorings that the corpus designers created by inlining existing methods and then using these expanded host methods to extract the same methods that were previously inlined. While these refactorings are realistic, they are not actual refactorings that open-source developers performed in their code. Thus, we use two other corpora that address this limitation.

(ii) *Community Corpus-B*: Silva et al. [57] maintain an active corpus containing 448 Java methods, each accompanied by its respective extract method refactorings that open-source developers actually performed. Importantly, this corpus spans all three categories of refactorings, as classified by Cossette and Walker [16]: automatable, non-automatable, and partially automatable by tools that try to replicate a real-world code change scenario. Non-automatable changes are those in which the developers combined refactorings with significant additional feature implementations (either in the host method or in the extracted method) in the same commit. Partially automatable changes are those in which developers made minor changes, such as renaming variables. Automatable changes represent “clean” commits in which the developers made a single type of change, i.e., performed *Extract Method* refactoring. We chose the latter category as our focus is on replicating the exact development scenarios where our tool only performs refactorings, it does not expand the code functionality. This resulted in 154 replicable refactorings.

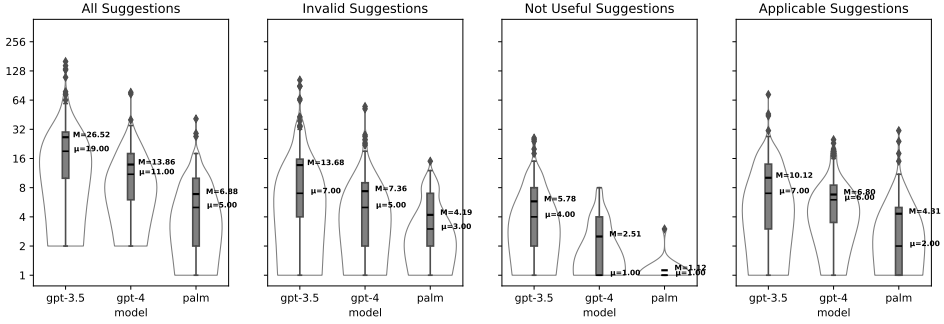


Fig. 3. The capabilities of LLMs in generating refactoring suggestions. The plots show the number of suggestions per host method (notice the exponential scale)

(iii) *Extended Corpus*: To enhance the robustness of our evaluation with a sizable oracle of actual refactorings performed by developers, we constructed Extended Corpus. To create it, we employed RefactoringMiner [62], 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 *Extract Method*. We ran it on highly regarded open-source repositories: *IntelliJ Community Edition* [33] and *CoreNLP* [32]. After filtering to remove refactoring commits that mixed feature additions (the one-liners and the extracted methods whose body overlapped a large proportion of the host method), we retain 2,849 *Extract Methods* from these repositories.

4.2 RQ1: Effectiveness of LLMs

Numerous LLMs have been developed by both open-source communities and proprietary companies [8, 23, 29, 41, 46, 53]. Among them, (i) PaLM [29], (ii) GPT-3.5 [12], and (iii) GPT-4 [46] 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 code refactoring tasks but are versatile, we want to determine their effectiveness in generating *Extract Method* refactoring suggestions.

4.2.1 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-A 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 3.1.2. 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 3.4), not useful suggestions (Definition 3.5), and applicable suggestions (Definition 3.6).

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.

4.2.2 Results. In Figure 3, 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 applicable 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 20, 11,

and 5 refactoring suggestions per host method, respectively. However, they also produced invalid suggestions with averages of 7, 5, and 3, respectively, and not useful suggestions with averages of 4, 1, and 1, respectively. Notably, GPT-3.5 yielded the highest number of applicable suggestions, with an average of 8 per host method, compared to 6 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 filter hallucinations and retain only applicable suggestions.

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 12,387 refactoring suggestions, of which 7,715 were hallucinations, leaving only 4,672 applicable suggestions. This further underscores the importance of employing post-processing techniques on LLM output, they cannot be used as-is for refactoring tasks.

Both GPT models are prolific at generating *Extract Method* suggestions, but more than half of the suggestions cannot be applied.

4.3 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 *Extract Method* 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.

4.3.1 Subject Systems and Experimental Setup. We use two oracle datasets, namely Community Corpus-A and Community Corpus-B. We focus on two key hyper-parameters: the number of iterations (I), with values in the set $\{1, 2, 3, \dots, 10\}$, and Temperature (T), for which we use the following values: $\{0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2\}$. For each combination (t, i) , where $t \in T$ and $i \in I$, we compute *EM-Assist*'s *Recall@5* to study how the tool's ability to suggest refactoring acceptable by humans varies with these settings. We use *Recall@5* as the target metric because while it is possible to collect the examples of real-world refactorings from the open-source projects, we cannot judge for sure if other tools' suggestions are incorrect. Thus, we aim to maximize the tool's recall while keeping the number of suggestions reasonably low.

We first define an experiment data point: for a given combination of LLM's Temperature/Iterations, we are replicating one of the refactoring scenarios from the dataset, *i.e.*, we are using *EM-Assist* to generate applicable suggestions for the given host method from the dataset. Following the practices used by other researchers [18, 56, 61, 66], for each experiment data point we evaluate the results using the top-5 applicable suggestions that *EM-Assist* produced. We compare these to the actual extracted method according to the oracle.

We count a match (or a hit) when a code fragment from any of the top-5 applicable suggestions matches the code fragment from the actual extracted method in the dataset, with an $n\%$ tolerance. For example, consider an extract method suggestion $e_i(n_i, (s_i, f_i))$ where s_i and f_i are the start and end line numbers of the suggested code fragment to be extracted, while the oracle specifies $e_j(n_j, (s_j, f_j))$ as the actual refactoring. The $n\%$ 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 $n\%$ of the length of the ground truth method. Formally, $\text{round}(s_j(1 - \text{len} \cdot n/100)) \leq s_i \leq \text{round}(s_j(1 + \text{len} \cdot n/100))$ and $\text{round}(f_j(1 - \text{len} \cdot n/100)) \leq f_i \leq \text{round}(f_j(1 + \text{len} \cdot n/100))$ where $\text{len} = f_j - s_j + 1$.

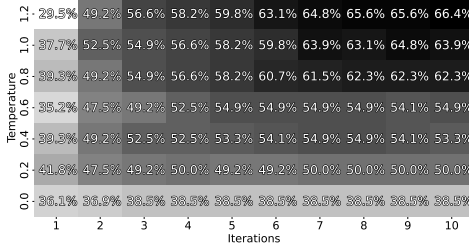


Fig. 4. Change of Recall@5 along with Temperature and iterations for the Community Corpus-A

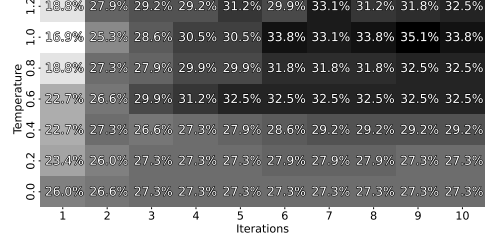


Fig. 5. Change of Recall@5 along with Temperature and iterations for the Community Corpus-B

Then, Recall@5 is the percentage of hits in top-5 at $n\%$ tolerance relative to the total number of actual extract methods in the oracle dataset. In other words, we compute the ratio of extract methods that we suggested and are perceived correctly by the oracle, while showing up to five suggestions per example.

We calculate Recall@5 using suggestions at 3% tolerance for each combination (t, i) on a version of *EM-Assist* with enhancements (see Section 3.4) turned on, and selecting the top-5 suggestions using the heat-popularity ranking $\mathcal{T}_3(e_i)$ (see Section 3.5). Additionally, we compute the metric for *EM-Assist*, with the enhancements turned off, and using the same heat-popularity ranking function $\mathcal{T}_3(e_i)$. This analysis not only aids in understanding the optimal settings for LLM but also sheds light on the impact of *EM-Assist*'s design decisions.

4.3.2 Results. Figure 4 and Figure 5 show via heatmaps how *EM-Assist*'s effectiveness varies with LLM Temperature/Iterations for the two corpora. The results show that higher temperature values and more iterations consistently yield superior results. For Community Corpus-A, *EM-Assist* achieved its highest Recall@5 values at 66.4%, with a temperature setting of 1.2 and 10 iterations. For Community Corpus-B, the best metric value was 35.1%, achieved with Temperature 1.0 and 9 iterations.

We also conducted a sensitivity analysis on the impact of *EM-Assist*'s design decision on improving its effectiveness. Table 1 shows an ablation study to find out the contributions of various modules from the *EM-Assist*'s pipeline (see Figure 2). For temperature 1.2 and 10 iterations, we compute Recall@5 after various stages in the *EM-Assist*'s pipeline, starting from *EM-Assist*'s raw performance with enhancements (*i.e.*, heuristics) and rankings disabled, all the way to activating all modules. The results show that *EM-Assist*'s design decisions contributed a maximum improvement of 27 percentage points for 1% tolerance. These results highlight the significant impact of *EM-Assist*'s design decisions.

With a higher temperature and number of iterations, LLMs produce better suggestions for *Extract Method* recommendation. Our ranking and heuristic techniques further rise the better suggestions to the top.

4.4 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.

4.4.1 Subject systems and Experimental Setup. We selected six tools that are representative of a wide array of techniques, including four (JDeodorant [61], JExtract [56], SEMI [14], and

Table 1. Ablation Study of *EM-Assist* showing the contributions of heuristics and ranking

Method	Recall@5		
	Tolerance 1%	Tolerance 2%	Tolerance 3%
Random rank, no heuristics	33.6%	35.2%	40.1%
Random rank, with heuristics	37.7%	39.3%	44.2%
<i>EM-Assist</i> - ($\mathcal{T}_1(e_i)$)	59.0%	59.8%	65.5%
<i>EM-Assist</i> - ($\mathcal{T}_2(e_i)$)	60.6%	61.4%	66.4%
<i>EM-Assist</i> - ($\mathcal{T}_3(e_i)$)	60.6%	61.4%	66.4%

Table 2. Evaluation results of *EM-Assist* with respect to six other tools on Community Corpus-A

Tool	Recall@5		
	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%
<i>EM-Assist</i> - ($\mathcal{T}_3(e_i)$)	60.6%	61.4%	66.4%

LiveRef [26]) that use static analysis-based rules and software quality metrics, and two (REMS [18] and GEMS [66]) that use prediction models based on machine learning techniques.

We first employ Community Corpus-A 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 Recall@5 at tolerance levels of 1%, 2%, and 3%, (for details on evaluation refer to Section 4.3.1). For *EM-Assist*, we compute the metric for the best-performing hyper-parameters, including the ranking function $\mathcal{T}_3(e_i)$, as identified in Section 4.3.1. For four tools (GEMS, JDeodorant, JExtract, SEMI), we reuse the evaluation results reported by other researchers [66] 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-A, 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 2,849 *Extract Method* refactorings that took place in open-source projects. For this extended evaluation, we compare with *LiveRef*, the most recent tool, due to its ability to be run automatically on extensive corpora, setting it apart from other available tools, which require manual interventions and thus cannot be run at scale.

4.4.2 Results. Table 2 shows the comparative effectiveness of *EM-Assist* in relation to the other six tools. Compared to the other evaluated tools, GEMS has the second best results with Recall@5 of 54.2%, 59.8%, and 62.6% at tolerances from 1% to 3%, respectively. *EM-Assist* achieved a higher Recall@5 across all tolerance values ranging from 60.6% to 66.4%, which outperforms the second-best tool.

To further strengthen the validity of our results, we applied *EM-Assist* on the Extended Corpus that includes 2,849 actual refactorings from open-source projects. We applied the most recent tool, *LiveRef*, to the same dataset. For both tools, we calculated the recall for top-5 suggestions with a 3% tolerance value. *LiveRef* achieved recall of 6.5%, **while *EM-Assist*'s recall was 42.1%, which**

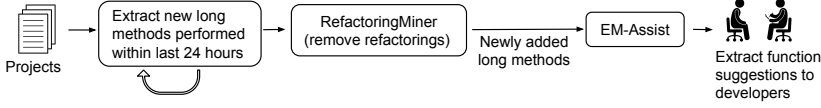


Fig. 6. The firehouse method we used to evaluate *EM-Assist*.

is **6.4x times higher**. These results show that *EM-Assist*’s effectiveness when applied to actual refactorings that developers perform in practice improves dramatically.

When using a workbench of synthetic refactorings, *EM-Assist* has a 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 6.4x.

4.5 RQ4: Usefulness of *EM-Assist*

EM-Assist is an interactive tool that provides maximum automation while still taking into account human input (see [7]). 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 the decision-making process underlying their acceptance or rejection. We will take into account this feedback when releasing future versions of *EM-Assist*.

4.5.1 Data and Experimental setup. To study the usefulness of the refactoring suggestions, we use the Firehouse survey research method [44]. 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 *Extract Method*. This immediate interaction allows developers to have fresh memories when responding to our questions, ensuring more reliable answers. Notably, this method, as demonstrated by Silva et al. [57], boasts a significantly higher response rate compared to other survey-based studies that require developer participation. Even more important, unlike previous studies that asked third-party developers to assess the quality of refactoring suggestions on arbitrary pieces of code they are not familiar with, we instead go directly to the developers of the code. Because they are the most familiar with that recent code, they are the most qualified to judge the quality of the suggestion.

We evaluate the usefulness of *EM-Assist*’s suggestions by conducting firehouse surveys with developers who work on *JetBrains/IntelliJ-Community Edition* and *JetBrains/Runtime*, both are open-source projects. IntelliJ Community Edition is written in Java and Kotlin, has 15.8k stars, 417k commits, and 952 contributors, showcasing its maturity and making it an ideal candidate for our study. JetBrainsRuntime is written in Java, has 969 stars, 76k commits, and 806 contributors.

Firehouse Survey Method: Figure 6 shows the process that we used for conducting firehouse surveys. We monitored the projects daily. We analyzed every new commit to identify newly added long methods. To enhance this process, we extended RefactoringMiner [62] to prevent mistakenly identifying refactored methods (e.g., that are 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).

After being presented with the top-3 refactoring suggestions generated by *EM-Assist* for their long method, 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.

Thematic Analysis: Since developers provide their motivations in free-form text, we employ thematic analysis, a widely used qualitative method for systematically extracting valuable information from their input. This approach, as recommended by Cruzes and Dybå [17], helps identify and report patterns (themes) within the data, facilitating its deeper understanding. We followed the steps required by thematic analysis as suggested by other researchers [11], in alignment with established research practices. Two authors of the paper independently read and assigned descriptive phrases (codes) to developers’ responses [11]. To ensure coding consistency between both labellers, they discussed and aligned their codes after labelling a random 25% sample of the data, as suggested by literature [13]. Subsequently, they coded the entire dataset of developers’ responses. After completing the process, the labellers finalized codes and derived thematic insights in line with established definitions of themes capturing data’s significant aspects.

4.5.2 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% [58], and this can be attributed to us using firehouse surveys. Table 3 shows the results, specifically the agreement levels. For each developer survey, we present the entry for the highest-rated suggestion by the developer. Table 3 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., choosing a positive rating). This shows *EM-Assist*’s ability to generate useful refactoring opportunities that developers agree with. Since *EM-Assist* 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.

Table 3. Developers’ levels of agreement to the refactoring suggestions produced by *EM-Assist*

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

Based on their qualitative comments, developers remarked that these refactoring suggestions significantly enhance code quality. They have also noted that *EM-Assist*’s suggestions provide fresh perspectives on their code, one developer stating “... *these suggestions made me look at this code with new eyes once more, and I will try to refactor it in a different way*”. Furthermore, developers express strong interest in seeing *EM-Assist* in production code, as exemplified by one developer’s gratitude: “*Thank you for interesting suggestions! Hope to see this in production in the future.*” These encouraging comments highlight the positive impact and potential usefulness of *EM-Assist*’s recommendations in the daily development process.

The top-3 reasons why developers did not accept *EM-Assist*’s suggestions are:

- (i) 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.
- (ii) The extracted method has too many parameters. This shows the importance of implementing additional filters based on the number of parameters in the suggestion.
- (iii) 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 a file, module, or project levels, for example, using techniques from [3, 4].

A detailed analysis of these qualitative aspects can be found in our companion materials [7].

Developers confirm that *EM-Assist*'s suggestions are useful, with 81% agreeing with them.

5 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 address this, we use an independent, publicly available dataset that is widely used by researchers to assess similar tools. Our rigorous evaluation of the tool on this dataset demonstrates its superior performance in terms of recall when compared to other state-of-the-art tools. Moreover, we use two other datasets containing *Extract Methods* that open-source developers performed in reputable repositories. 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. 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 programming languages.

(3) **Verifiability:** We make the *EM-Assist*, the datasets and results, publicly available [7].

6 RELATED WORK

We group the related work into four categories: (i) static analysis and (ii) machine learning techniques for *Extract Method* refactoring, (iii) various studies on *Extract Method* refactoring, and (iv) tools that employ LLMs for various SE tasks.

Static analysis: We present more details about the four static analysis tools for which we conduct a direct comparison in our evaluation (see Section 4.4). JDeodorant [61] 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 [56] 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 [14] explores the coherence between statements and returns code fragments with high cohesion as refactoring candidates. LiveRef [26] is an IntelliJ plugin that offers real-time *Extract Method* 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.

Other researchers suggest *Extract Method* refactoring based on program slicing [1, 35, 39]. Others used software design principles, such as *separation of concerns* [55], or *single responsibility principle* [14]. Several others [4, 59, 69, 70] suggest *Extract Method* refactoring on the basis of eliminating code clones and code duplication. In contrast, we are using static analysis techniques to check and enhance the LLM results and to compensate for LLMs lacking program semantics for executing *Extract Method* correctly. Similar to [14, 25, 56, 66, 67] we employ a ranking mechanism for *EM-Assist*'s suggestions. Whereas previous work uses software quality metrics such as cyclomatic

complexity, nesting depth and length, in our approach we rely on LLM-specific quality metrics, such as popularity and heat.

ML Models: We present more details about the two ML-based tools for which we conduct a direct comparison in our evaluation (see Section 4.4). REMS [18] utilizes multi-view representations from the code property graph. It then trains a machine learning classifier to guide the extraction of suitable lines of code as a new method. GEMS [66] encodes metrics related to complexity, cohesion, and coupling as features to train machine learning classifiers for recommending Extract Method refactoring opportunities.

Other researchers worked on discovering the need for applying *Extract Method* refactoring. Aniche et al. [6] evaluated the effectiveness of various ML algorithms for predicting software refactorings, including *Extract Method*. The models were trained on thousands of refactorings mined with RefactoringMiner [62] from open-source projects. Van der Leij et al. [63] replicated the study [6] at the ING company and discovered that ML models predict very well the opportunity for applying *Extract Method* refactoring. Others [5, 51] use commit messages to predict software refactorings. While these techniques 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.

Miscellaneous *Extract Method* studies: Several studies [43, 45, 62] show that *Extract Method* refactoring is among the top-5 most frequently performed in practice, both for manual and automated refactoring. Murphy-Hill and Black [42] were some of the first researchers to uncover challenges that developers face when performing *Extract Method* refactorings.

LLMs in SE tasks: Hou et al. [31] conducted a comprehensive literature review on the application of LLMs in SE. White et al. [65] introduce patterns for prompting LLMs for various SE tasks, including code refactoring. In contrast, our work presents empirically-justified best practices for using LLMs for refactoring tasks.

7 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 *Extract Method* 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.

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.

In this work, we focused on *Extract Method* refactoring for Java and Kotlin, but our approach can be expanded to other refactoring automation tasks, and to other programming languages. 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.

8 DATA AVAILABILITY

We have made our tool and evaluation data publicly available on our website [7].

REFERENCES

- [1] Aharon Abadi, R Ettinger, and YA Feldman. 2009. Fine slicing for advanced method extraction. In *3rd workshop on refactoring tools*, Vol. 21.
- [2] Eliseeva Aleksandra, Sokolov Yaroslav, Bogomolovand Egor, Golubev Yaroslav, Dig Danny, and Bryksin Timofey. ASE 2023. From Commit Message Generation to History-Aware Commit Message Completion. (ASE 2023). <https://arxiv.org/pdf/2308.07655.pdf>
- [3] Eman Alomar, Anton Ivanov, Zarina Kurbatova, Yaroslav Golubev, Mohamed Wiem Mkaouer, Ali Ouni, Timofey Bryksin, Le Nguyen, Amit Kini, and Aditya Thakur. 2023. AntiCopyPaster: Extracting Code Duplicates As Soon As They Are Introduced in the IDE. 1–4. <https://doi.org/10.1145/3551349.3559537>
- [4] Eman Alomar, Anton Ivanov, Zarina Kurbatova, Yaroslav Golubev, Mohamed Wiem Mkaouer, Ali Ouni, Timofey Bryksin, Le Nguyen, Amit Kini, and Aditya Thakur. 2023. Just-in-time code duplicates extraction. *Information and Software Technology* 158 (02 2023), 107169. <https://doi.org/10.1016/j.infsof.2023.107169>
- [5] Eman Abdullah AlOmar, Jiaqian Liu, Kenneth Addo, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Zhe Yu. 2022. On the documentation of refactoring types. *Automated Software Engineering* 29 (2022), 1–40.
- [6] Mauricio Aniche, Erick Maziero, Rafael Durelli, and Vinicius HS Durelli. 2020. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering* 48, 4 (2020), 1432–1450.
- [7] Anonymous. 2023. *Refactoring Package at GitHub*. <https://llm-refactoring.github.io/>
- [8] Anthropic. 2023. Introducing Claude. <https://www.anthropic.com/index/introducing-claude>
- [9] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology* 108 (2019), 115–138.
- [10] Rajiv D Banker, Srikant M Datar, Chris F Kemerer, and Dani Zweig. 1993. Software complexity and maintenance costs. *Commun. ACM* 36, 11 (1993), 81–95.
- [11] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101. <https://doi.org/10.1191/1478088706qp0630a> arXiv:<https://www.tandfonline.com/doi/pdf/10.1191/1478088706qp0630a>
- [12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf
- [13] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen. 2013. Coding In-depth Semistructured Interviews: Problems of Unitization and Inter-coder Reliability and Agreement. *Sociological Methods & Research* 42, 3 (2013), 294–320. <https://doi.org/10.1177/0049124113500475>
- [14] Sofia Charalampidou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Antonios Gkortzis, and Paris Avgeriou. 2017. Identifying Extract Method Refactoring Opportunities Based on Functional Relevance. *IEEE Transactions on Software Engineering* 43, 10 (2017), 954–974. <https://doi.org/10.1109/TSE.2016.2645572>
- [15] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2022. An Empirical Study on the Usage of Transformer Models for Code Completion. *IEEE Transactions on Software Engineering* 48, 12 (2022), 4818–4837. <https://doi.org/10.1109/TSE.2021.3128234>
- [16] Bradley E. Cossette and Robert J. Walker. 2012. Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries (FSE ’12). Association for Computing Machinery, New York, NY, USA, Article 55, 11 pages. <https://doi.org/10.1145/2393596.2393661>
- [17] Daniela S. Cruzes and Tore Dybå. 2011. Research synthesis in software engineering: A tertiary study. *Information and Software Technology* 53, 5 (2011), 440–455. <https://doi.org/10.1016/j.infsof.2011.01.004> Special Section on Best Papers from XP2010.
- [18] Di Cui, Qiangqiang Wang, Siqi Wang, Jianlei Chi, Jianan Li, Lu Wang, and Qingshan Li. 2023. 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)*. 191–202. <https://doi.org/10.1109/ICPC58990.2023.00034>
- [19] Malinda Dilhara, Abhiram Bellur, Danny Dig, and Timofey Bryksin. 2024. Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example. In *32nd ACM Symposium on the Foundations of Software Engineering (FSE ’24)*. to appear.
- [20] Malinda Dilhara, Ameya Ketkar, and Danny Dig. 2021. Understanding Software-2.0: A Study of Machine Learning Library Usage and Evolution. *ACM Trans. Softw. Eng. Methodol.* 30, 4, Article 55 (jul 2021), 42 pages. <https://doi.org/10.1145/3453478>

- [21] Malinda Dilhara, Ameya Ketkar, Nikhith Sannidhi, and Danny Dig. 2022. Discovering repetitive code changes in python ML systems. In *Proceedings of the 44th International Conference on Software Engineering (Pittsburgh, Pennsylvania) (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 736–748. <https://doi.org/10.1145/3510003.3510225>
- [22] Sarah Fakhoury, Devjeet Roy, Adnan Hassan, and Venera Arnaoudova. 2019. Improving source code readability: Theory and practice. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2–12.
- [23] Falcon. 2023. Falcon. <https://falconllm.tii.ae>
- [24] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
- [25] Sara Fernandes, Ademar Aguiar, and André Restivo. 2022. A Live Environment to Improve the Refactoring Experience. In *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming*. 30–37.
- [26] Sara Fernandes, Ademar Aguiar, and André Restivo. 2023. LiveRef: A Tool for Live Refactoring Java Code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (Rochester, MI, USA) (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 161, 4 pages. <https://doi.org/10.1145/3551349.3559532>
- [27] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
- [28] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R. Lyu. ASE 2023. 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 (ASE '38)*. ACM. <https://arxiv.org/abs/2304.07575>
- [29] GoogleAI. 2023. Google Bard: An Early Experiment with Generative AI. <https://ai.google/static/documents/google-about-bard.pdf>
- [30] Roman Haas and Benjamin Hummel. 2015. Deriving extract method refactoring suggestions for long methods. In *International Conference on Software Quality*. Springer, 144–155.
- [31] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. *arXiv preprint arXiv:2308.10620* (2023).
- [32] JetBrains. 2023. CoreNLP. (2023). <https://github.com/stanfordnlp/CoreNLP>
- [33] JetBrains. 2023. IntelliJ Community Edition. (2023). <https://github.com/JetBrains/intellij-community>
- [34] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. *arXiv:2211.15533* [cs.CL]
- [35] Arun Lakhotia and Jean-Christophe Deprez. 1998. Restructuring programs by tucking statements into functions. *Information and Software Technology* 40, 11-12 (1998), 677–689.
- [36] Yinhan Liu, MyLe Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2020. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692* (2020).
- [37] Dung Nguyen Manh, Nam Le Hai, Anh T. V. Dau, Anh Minh Nguyen, Khanh Nghiem, Jin Guo, and Nghi D. Q. Bui. 2023. The Vault: A Comprehensive Multilingual Dataset for Advancing Code Understanding and Generation. *arXiv:2305.06156* [cs.CL]
- [38] Robert C. Martin. 2017. *Clean Architecture: A Craftsman’s Guide to Software Structure and Design* (1st ed.). Prentice Hall Press, USA.
- [39] Katsuhisa Maruyama. 2001. Automated Method-Extraction Refactoring by Using Block-Based Slicing. In *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context (Toronto, Ontario, Canada) (SSR '01)*. Association for Computing Machinery, New York, NY, USA, 31–40. <https://doi.org/10.1145/375212.375233>
- [40] Panita Meananeatra, Songsakdi Rongviriyapanish, and Taweessup Apiwattanapong. 2018. Refactoring opportunity identification methodology for removing long method smells and improving code analyzability. *IEICE TRANSACTIONS on Information and Systems* 101, 7 (2018), 1766–1779.
- [41] Meta. 2023. Introducing Llama. <https://ai.meta.com/llama/>
- [42] Emerson Murphy-Hill and Andrew P Black. 2008. Breaking the barriers to successful refactoring: observations and tools for extract method. In *Proceedings of the 30th international conference on Software engineering*. 421–430.
- [43] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (2012), 5–18. <https://doi.org/10.1109/TSE.2011.41>
- [44] Emerson Murphy-Hill, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan. 2015. The Design Space of Bug Fixes and How Developers Navigate It. *IEEE Transactions on Software Engineering* 41, 1 (2015), 65–81. <https://doi.org/10.1109/TSE.2014.2357438>

- [45] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. 2013. A Comparative Study of Manual and Automated Refactorings. In *ECOOP 2013 – Object-Oriented Programming*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 552–576.
- [46] OpenAI. 2023. GPT-4 Technical Report. (2023). <https://arxiv.org/pdf/2303.08774.pdf>
- [47] Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334* (2023).
- [48] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
- [49] Daniel Ramos, Hailie Mitchell, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. ASE 2023. MELT: Mining Effective Lightweight Transformations from Pull Requests. (ASE 2023). <https://arxiv.org/abs/2308.14687>
- [50] Laria Reynolds and Kyle McDonell. 2021. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) (CHI EA '21). Association for Computing Machinery, New York, NY, USA, Article 314, 7 pages. <https://doi.org/10.1145/3411763.3451760>
- [51] Priyadarshni Suresh Sagar, Eman Abdulah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Christian D Newman. 2021. Comparing commit messages and source code metrics for the prediction refactoring activities. *Algorithms* 14, 10 (2021), 289.
- [52] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. 2019. Automatically assessing code understandability. *IEEE Transactions on Software Engineering* 47, 3 (2019), 595–613.
- [53] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Galle, et al. 2022. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100* (2022).
- [54] Feng Sidong and Chen Chunyang. 2024. Prompting Is All Your Need: Automated Android Bug Replay with Large Language Models. In *Proceedings of the 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 10 pages.
- [55] Danilo Silva, Ricardo Terra, and Marco Tulio Valente. 2014. Recommending automated extract method refactorings. In *Proceedings of the 22nd International Conference on Program Comprehension*. 146–156.
- [56] Danilo Silva, Ricardo Terra, and Marco Túlio Valente. 2015. Jextract: An eclipse plug-in for recommending automated extract method refactorings. *arXiv preprint arXiv:1506.06086* (2015).
- [57] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors (FSE 2016). Association for Computing Machinery, New York, NY, USA, 858–870. <https://doi.org/10.1145/2950290.2950305>
- [58] Janice Singer, Susan E. Sim, and Timothy C. Lethbridge. 2008. *Software Engineering Data Collection for Field Studies*. Springer London, London, 9–34. https://doi.org/10.1007/978-1-84800-044-5_1
- [59] Robert Tairas and Jeff Gray. 2012. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Information and Software Technology* 54, 12 (2012), 1297–1307.
- [60] Omkarendra Tiwari and Rushikesh Joshi. 2022. Identifying Extract Method Refactorings. In *15th Innovations in Software Engineering Conference (Gandhinagar, India) (ISEC 2022)*. Association for Computing Machinery, New York, NY, USA, Article 7, 11 pages. <https://doi.org/10.1145/3511430.3511435>
- [61] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2011. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software* 84, 10 (2011), 1757–1782. <https://doi.org/10.1016/j.jss.2011.05.016>
- [62] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2022), 930–950. <https://doi.org/10.1109/TSE.2020.3007722>
- [63] David van der Leij, Jasper Binda, Robbert van Dalen, Pieter Vallen, Yaping Luo, and Mauricio Aniche. 2021. 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*. 1337–1347.
- [64] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, Vol. 35. Curran Associates, Inc., 24824–24837. https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf
- [65] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *arXiv preprint arXiv:2303.07839* (2023).
- [66] Sihan Xu, Aishwarya Sivaraman, Siau-Cheng Khoo, and Jing Xu. 2017. GEMS: An Extract Method Refactoring Recommender. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. 24–34. <https://doi.org/10.1109/ISSRE.2017.8006001>

[//doi.org/10.1109/ISSRE.2017.35](https://doi.org/10.1109/ISSRE.2017.35)

- [67] Limei Yang, Hui Liu, and Zhendong Niu. 2009. Identifying Fragments to Be Extracted from Long Methods. In *Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference (APSEC '09)*. IEEE Computer Society, USA, 43–49. <https://doi.org/10.1109/APSEC.2009.20>
- [68] Limei Yang, Hui Liu, and Zhendong Niu. 2009. Identifying fragments to be extracted from long methods. In *2009 16th Asia-Pacific Software Engineering Conference*. IEEE, 43–49.
- [69] Norihiro Yoshida, Seiya Numata, Eunjong Choiz, and Katsuro Inoue. 2019. Proactive clone recommendation system for extract method refactoring. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWor)*. IEEE, 67–70.
- [70] Ruru Yue, Zhe Gao, Na Meng, Yingfei Xiong, Xiaoyin Wang, and J David Morgenthaler. 2018. Automatic clone recommendation for refactoring based on the present and the past. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 115–126.