Together We Are Better: LLM, IDE and Semantic Embedding to Assist Move Method Refactoring

FRAOL BATOLE, Tulane University, USA
ABHIRAM BELLUR, University of Colorado Boulder, USA
MALINDA DILHARA, University of Colorado Boulder, USA
YAROSLAV ZHAROV, JetBrains Research, Germany
TIMOFEY BRYKSIN, JetBrains Research, Cyprus
KAI ISHIKAWA, NEC, Japan
HAIFENG CHEN, NEC, USA
MASAHARU MORIMOTO, NEC, USA
MOTOURA SHOTA, NEC, USA
TAKEO HOSOMI, NEC, USA
TIEN N. NGUYEN, University of Texas at Dallas, USA
HRIDESH RAJAN, Tulane University, USA
NIKOLAOS TSANTALIS, Concordia University, Canada
DANNY DIG, JetBrains Research, University of Colorado Boulder, USA

MoveMethod is a hallmark refactoring to remedy the lack of code modularity and remove several code smells that contribute to technical debt. Despite a plethora of research tools that recommend which methods to move and where by optimizing software quality metrics, these recommendations do not align with how expert developers perform MoveMethod. We hypothesize that given the huge training of Large Language Models and their reliance upon the *naturalness of code*, they should be better at recommending which methods are *misplaced* in a given class and which classes are better hosts for such misplaced methods. Moreover, their recommendations should better align with experts. Our formative study of 2016 LLM recommendations revealed that LLMs give expert suggestions, yet they are unreliable: up to 80% of the suggestions are hallucinations.

We introduce the first LLM-powered assistant for MoveMethod refactoring that automates its whole end-to-end lifecycle, from recommendation to execution. We designed novel solutions to overcome the limitations of LLM-based MoveMethod refactoring. We automatically filter LLM hallucinations using static analysis from IDEs and a novel workflow that requires LLMs to be *self-consistent, critique, and rank* refactoring suggestions. Moreover, MoveMethod refactoring requires global, project-level reasoning to determine the best target

Authors' addresses: Fraol Batole, fbatole@tulane.edu, Tulane University, New Orleans, Louisiana, USA; Abhiram Bellur, abhiram.bellur@colorado.edu, University of Colorado Boulder, Boulder, Colorado, USA; Malinda Dilhara, malinda.malwala@colorado.edu, University of Colorado Boulder, Boulder, Colorado, USA; Yaroslav Zharov, yaroslav.zharov@jetbrains.com, JetBrains Research, , , Germany; Timofey Bryksin, timofey.bryksin@jetbrains.com, JetBrains Research, , , Cyprus; Kai Ishikawa, k-ishikawa@nec.com, NEC, , , Japan; Haifeng Chen, haifeng@nec-labs.com, NEC, , , USA; Masaharu Morimoto, m-morimoto@nec.com, NEC, , , USA; Motoura Shota, motoura@nec.com, NEC, , , USA; Takeo Hosomi, takeo.hosomi@nec.com, NEC, , , USA; Tien N. Nguyen, tien.n.nguyen@utdallas.edu, University of Texas at Dallas, Dallas, Texas, USA; Hridesh Rajan, hrajan@tulane.edu, Tulane University, New Orleans, Louisiana, USA; Nikolaos Tsantalis, nikolaos.tsantalis@concordia.ca, Concordia University, , , Canada; Danny Dig, danny.dig@jetbrains.com, JetBrains Research, University of Colorado Boulder, Boulder, Colorado, USA.

ACM XXXX-XXXX/2025/1-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

classes where to relocate a misplaced method. We solved the limited context size of LLMs by employing *refactoring-aware retrieval augment generation* (RAG). We implemented our approach as an IntelliJ IDEA plugin, MM-Assist, that works for Java code. It synergistically combines the strengths of the LLM, IDE, static analysis, and semantic relevance. MM-Assist generates candidates, filters LLM hallucinations, validates and ranks recommendations, and then finally executes the correct refactoring based on user approval. In our thorough, multi- methodology empirical evaluation, we compare MM-Assist with the previous state-of-the-art approaches. MM-Assist significantly outperforms them: on a benchmark widely used by other researchers, our Recall@1 and Recall@3 are 73% and 80%, respectively, which is a 2x improvement over previous state-of-the-art approaches (33% and 37%). Moreover, we extend the corpus used by previous researchers with 210 actual refactorings performed by Open-source software developers in 2024; MM-Assist achieves even more significant improvements over previous tools, our Recall@1 is 71%, and Recall@3 is 82%, compared to 20% for feTruth– this is an almost 4x improvement. Lastly, we conducted a user case study with 30 experienced participants who used MM-Assist to refactor their own code for one week. They rated 82.8% of MM-Assist recommendations positively. This shows that MM-Assist is both effective and useful.

ACM Reference Format:
Fraol Batole, Abhiram Bellur, Malinda Dilhara, Yaroslav Zharov, Timofey Bryksin, Kai Ishikawa, Haifeng Chen, Masaharu Morimoto, Motoura Shota, Takeo Hosomi, Tien N. Nguyen, Hridesh Rajan, Nikolaos Tsantalis, and Danny Dig. 2025. Together We Are Better: LLM, IDE and Semantic Embedding to Assist Move Method Refactoring. 1, 1 (January 2025), 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

MoveMethod is a key refactoring technique [20] that moves a misplaced method from its original class (source) to a more suitable target class. A method is considered *misplaced* when it doesn't interact with its own class's state or relies more on the state of another class. Developers use MoveMethod to improve modularity by grouping related behavior (methods) with the data (fields) it affects, which enhances class cohesion and reduces coupling between classes. MoveMethod helps eliminate several code smells, such as FeatureEnvy [49], GodClass [4], DuplicatedCode [28], and MessageChain [20]. Thus, MoveMethod reduces technical debt and is one of the top-5 most common refactorings [35, 37, 50], in both manual and automated settings.

The MoveMethod lifecycle comprises four phases: (i) identifying a misplaced method `m` in its host class `H`, (ii) finding a more suitable target class `T`, (iii) ensuring refactoring pre- and post-conditions to preserve program behavior, and (iv) executing the refactoring mechanics (i.e., source code transformation). Each phase presents challenges. Identifying candidates requires a solid grasp of design principles and the entire codebase, while checking preconditions [39, 49] involves complex static analysis. The mechanics also pose difficulties, such as relocating `m`, updating its call sites, and adjusting field and method accesses. Due to such complexities, most existing solutions don't provide an end-to-end solution; IDEs focus on preconditions and mechanics, while research tools mainly identify refactoring opportunities.

The research community has proposed various approaches [5, 6, 11, 24, 29, 30, 32, 47, 49] for recommending misplaced methods or new target classes, typically optimizing software quality metrics like cohesion and coupling. These approaches fall into three categories: (i) static analysis-based [5, 47, 49], (ii) machine learning classifiers [6, 11, 24], and (iii) deep learning-based [29, 30, 32]. However, static analysis approaches rely on expert-defined thresholds, and ML/DL methods need constant retraining as coding standards and practices evolve. Additionally, their recommendations often don't align with how expert developers perform MoveMethod in practice.

We hypothesize that achieving good software design that is easy to understand and resilient to future changes is a *balancing act between science* (e.g., metrics, design principles) and *art* (e.g., experience, expertise, and intuition about what constitute good abstractions). This can explain why code optimized via software quality metrics is not always accepted in practice [15–17, 19, 23, 45].

In this paper, we introduce the first approach to automate the entire MoveMethod refactoring lifecycle using Large Language Models (LLMs). We hypothesize that, due to their extensive pre-training on billions of methods and their reliance on the *naturalness of code*, LLMs can generate an abundance of MoveMethod recommendations. We also expect LLM recommendations to better align with expert practices. We use GPT-4o, which has demonstrated superior performance on programming tasks [1, 8] and is widely adopted in developer productivity tools [12, 21]. In our formative study, we found LLMs are prolific in generating suggestions, averaging 6 recommendations per class. However, two major challenges must be addressed to make this approach practical.

First, LLMs can produce *hallucinations*, i.e., recommendations that seem plausible but are flawed. In our formative study of 2016 LLM recommendations, we identified three types of hallucinations: (i) LLM suggests target classes that do not exist in the project, (ii) it is impossible to move a method in the suggested target class, and (iii) LLM identifies invalid methods as misplaced in a host class. Our findings reveal that up to 80% of LLM recommendations are hallucinations. This requires further processing to enhance LLM reliability.

We discovered novel ways to automatically eliminate LLM hallucinations. We complement LLM reasoning (i.e., *the creative, non-deterministic, and artistic part* akin to human naturalness) with static analysis embedded in the IDE (i.e., *the rigorous, deterministic, scientific part*) to achieve better results. Mature refactoring implementations in existing IDEs like IntelliJ IDEA contain a plethora of static analysis checks (called refactoring preconditions [39]) that must be true before a method can be safely moved to another class. We leverage these refactoring preconditions to validate LLM recommendations. Moreover, we compute semantic similarity between each method and its host class using code-trained embeddings, identifying the least cohesive methods and focus the LLM on these methods. These stages effectively removed *all* LLM hallucinations. We present these techniques in Section 3.2.

Second, MoveMethod refactoring requires global, project-level reasoning to determine the best target classes where to relocate a misplaced method. However, passing an entire project in the prompt is beyond the limited window size of current LLMs [52]. Even with future versions of LLMs that continuously increase the window size, passing the whole project as context introduces noise and redundancy, as not all classes are relevant; instead this further distracts the LLM [36, 52].

We address the limited context size of LLMs by using *retrieval augmented generation* (RAG) to enhance the LLM's input with relevant project-specific information for better decision-making. However, a naive approach of retrieving all similar classes would be ineffective and worsen the hallucination problem. Instead, we first apply IDE-based static analysis to identify mechanically feasible target classes, significantly narrowing the search space and reducing hallucinations. While static analysis ensures feasibility, we also leverage semantic relevance to find a suitable target class. We utilize VoyageAI [53], which has demonstrated state-of-the-art performance in code-related tasks [54]. This two-step process combines mechanical feasibility with semantic relevance, enabling our approach to make informed decisions and perform global project-level reasoning. We coin this approach *refactoring-aware retrieval augmented generation*, which addresses LLM hallucinations and context limitations while fulfilling the specific needs of MoveMethod refactoring (see Section 3.3).

We designed, implemented, and evaluated these novel solutions as an IntelliJ IDEA plugin for Java code, MM-Assist. It synergistically combines the strengths of the LLM, IDE, static analysis, and semantic relevance. MM-Assist generates candidates, filters LLM hallucinations, validates and ranks recommendations, and then finally executes the correct refactoring based on user approval.

We designed a comprehensive, multi-methodology evaluation of MM-Assist to corroborate, complement and expand research findings: formative study, comparative study, replication of real-world refactorings, repository mining, user/case study, and questionnaire surveys. Among others, our formative study of 2016 LLM recommendations reveal the strengths and weaknesses

of using LLMs for recommending MoveMethod refactorings. To quantify the improvements of MM-Assist over the vanilla LLM solution, we conducted an ablation study that shows MM-Assist brings significant improvements. Moreover, we compare MM-Assist with the previous best in class approaches in their respective markets: JMove [47], which uses static analysis, and feTruth [29], which uses Deep Learning; these have been shown previously to outperform all previous MoveMethod recommendation tools. Using a synthetic corpus widely used by previous approaches, we found that MM-Assist significantly outperforms them: for instance methods, our Recall@1 and Recall@3 are 73% and 80%, respectively, which is an almost double improvement over previous state-of-the-art approaches (40% and 42%). Moreover, we extend the corpus used by previous researchers with 210 actual refactorings that we mined from OSS repositories in 2024 (thus avoiding LLM data contamination), containing both instance and Java static methods. We compared against JMove and feTruth on this real-world oracle, and found that MM-Assist significantly outperforms previous tools. Our Recall@1 is 71%, and Recall@3 is 82%, compared to the previous best tool, feTruth, which achieved a Recall@3 of 20% – this is a 4x improvement. This shows that MM-Assist's recommendations better align with human developer best practices.

To recommend which method(s) to move from a class, previous tools require analyzing an entire project – which takes between 2 hours for medium projects to more than 10 hours for 1M LOC projects, and they overwhelm the user with up to 40 recommendations per class. MM-Assist is modular: it takes on average 30 seconds (even on projects with tens of thousands of classes), and it shows no more than 3 high quality recommendations per class. Thus, MM-Assist is practical.

To better understand how developers use MM-Assist in practice, we recruited 30 experienced participants who used it on their own code for a week and provided telemetry data. Unlike previous studies that had participants assess recommendations on unfamiliar third-party code, our study allows participants to evaluate recommendations on code they deeply understand. Results show that 82.8% of participants rated MM-Assist's recommendations positively and preferred our LLM-based approach over classic IDE workflows. One participant remarked, *"I am fairly skeptical when it comes to AI in my workflow, but still excited at the opportunity to delegate grunt work to them."*

This paper makes the following contributions:

- **Approach.** We present the first LLM-powered assistant that supports the lifecycle of recommending and applying MoveMethod refactorings. Our tool offers key advantages: (i) it ensures the recommendations are feasible and executes them correctly upon user approval, (ii) it requires no user-specified thresholds or model (re)-training, making it more future-proof as LLMs evolve, and (iii) it handles both instance methods — like other solutions — and static methods, which others avoid due to the large search space.
- **Best Practices.** We discovered a new set of best practices to overcome the LLM limitations when it comes to refactorings that require global reasoning. We automatically filter LLM hallucinations and conquer the LLM's limited context size using refactoring-aware RAG.
- **Implementation.** We designed, implemented, and evaluated these ideas in an IntelliJ plugin, MM-Assist, that works on Java code. It addresses practical considerations for tools used in the daily workflow of software developers.
- **Evaluation.** We thoroughly evaluated MM-Assist, and it outperforms previous best-in-class approaches. We also created an oracle replicating actual refactorings done by OSS developers, where MM-Assist showed even more improvements. A user study confirms that developers prefer our LLM-based assistant, demonstrating that MM-Assist aligns with and replicates real-world expert logic.

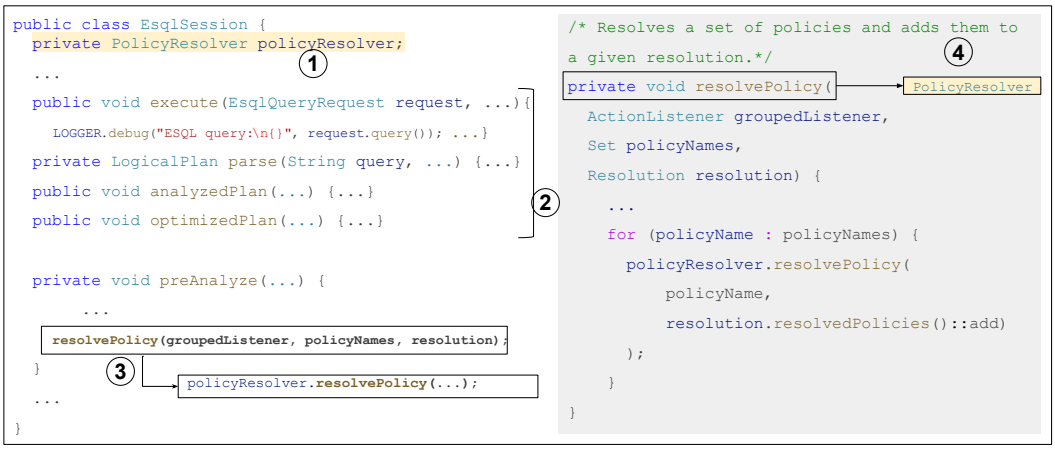

Fig. 1. A real-world example demonstrating a MoveMethod on `resolvePolicy` performed by developers in the *Elasticsearch* project, commit 876e7015

2 MOTIVATING EXAMPLE

We illustrate the challenges of recommending MoveMethod using a real-world refactoring that occurred in the *Elasticsearch* project – a distributed open-source search and analytics engine. We illustrate the refactoring in Figure 1, and the full commit can be seen in [18]. The `resolvePolicy` method (See ④ in Figure 1), originally part of the `EsqlSession` class, is misplaced. While `EsqlSession` handles parsing and executing queries, `resolvePolicy` is responsible for resolving and updating policies. Specifically, `resolvePolicy` accesses the field `policyResolver` (See ①) and parameters like `groupedListener`, `policyNames`, and `resolution`. Recognizing this misalignment, the developers refactored the code by moving `resolvePolicy` to the `PolicyResolver` class (not shown in the figure due to space constraints), updating the method body accordingly, and modifying the call sites (See ③). After the refactoring, both `EsqlSession` and `PolicyResolver` became more cohesive.

Automating the identification of such refactoring opportunities is essential for maintaining software quality, but it poses significant challenges for existing tools. We first applied the state-of-the-art Feature Envy detection technique, feTruth [29], to this scenario. However, feTruth failed to recommend the actual refactoring performed by the developers. Instead, it suggested moving `fieldNames(...)`, which is a method highly cohesive with the class's primary responsibilities. Since feTruth relies on training DL models for refactoring tasks using real-world datasets, this result highlights that even specialized DL models may not capture all refactoring scenarios where domain-specific knowledge or project-specific design patterns play key roles.

Next, we ran JMove [47], a state-of-the-art MoveMethod recommendation tool that solely relies on static analysis. To analyze the whole project JMove requires more than 12 hours. To speed up JMove, we ran it on a sub-project of Elasticsearch containing `EsqlSession`. Unfortunately, JMove did not produce any recommendations for the `EsqlSession` class. Furthermore, JMove took 30 minutes to report any results as it needed to analyze the entire sub-project and create program dependencies. This illustrates another major shortcoming of previous tools like JMove as developers would run out of patience when executing tools on medium to large size projects like `Elasticsearch`, which has 800K LOC.

We then explore the potential of Large Language Models (LLMs) to recommend MoveMethod refactoring. We used GPT-4o, a state-of-the-art LLM developed by OpenAI [7], and prompted

it with the content of the `EsqlSession` class, asking: *"Identify the method that should move out of the* `EsqlSession` *class and where to move it."* Our experiment highlighted both the strengths and limitations of LLMs for this task. In order of priority, the LLM identified 5 methods for relocation (see ②), including `execute`, `parse`, `optimizedPlan`, and `analysePlan`, all of which rightly belong in `Esqlsession` and were never moved by the developers. However, the LLM did successfully identify `resolvePolicy` as a candidate for refactoring, showing its ability to detect semantically *misplaced* methods. Despite this success, the LLM recommended other methods before `resolvePolicy`, meaning that a developer would need to filter out several irrelevant suggestions before arriving at the correct one.

After identifying that the method `resolvePolicy` is misplaced, a tool must find a suitable target class to move the method into. While the LLM was able to recommend the correct target class, it also responded with (i) two target classes (i.e., `Resolution`, `ActionListener`), which are plausible target classes, but are not the best semantically fit for the method; (ii) two hallucinations, i.e., classes that do not exist (i.e., `PolicyResolutionService`, `PolicyUtils` as the LLM lacks project-wide context.

A naive approach to address the LLM's lack of project-wide understanding is to prompt it with the entire codebase. However, this is currently impractical due to the LLM's context size limitations and inability to efficiently handle long contexts [33]. Even state-of-the-art LLMs can't process large projects like *Elasticsearch* in a single prompt without truncating crucial information. Moreover, as context capacities expand, processing an entire project at once remains challenging due to increased computational complexity, memory demands, and reduced reasoning effectiveness [33]. Additionally, the cost of processing such large inputs with commercial LLM APIs is prohibitive.

These experiments reveal both the strengths and limitations of LLMs for MoveMethod refactoring. On the positive side, LLMs show proficiency in generating multiple suggestions and demonstrate an ability to identify methods that are semantically misplaced. However, they also exhibit significant limitations, including difficulty in suggesting appropriate target classes, and a high rate of irrelevant or infeasible suggestions. These limitations underscore the need for caution when relying on LLM-generated refactoring recommendations. Developers attempting to leverage LLMs for refactoring would face a laborious and error-prone process. They would need to manually collect and re-analyze the suggested methods, verify the suitability of each method for relocation, prompt the LLM again for target class suggestions for suitable methods, and meticulously identify and filter out hallucinations such as non-existent classes and methods that are impossible to move. In the example (Figure 1), a developer would need to sift through 5 candidate methods and, for each method, understand if any of the 5 or more proposed target classes are adequate. The developers would analyze 20 or more (method, targetClass) pairs before finding one they agree with.

The above example motivates our approach, MM-Assist, which significantly streamlines the refactoring process by (1) utilizing semantic relevance to collect a list of target methods that are the least compatible with the host class, (2) employing static analysis to validate and filter suggestions, and (3) leveraging LLMs to prioritize only valid recommendations. For the example above, MM-Assist was able to recommend moving `resolvePolicy` to the `PolicyResolver` class as the top candidate. Our tool, MM-Assist, liberates the developers so they can focus on the creative part. Rather than sifting through numerous invalid, non-existent, or impractical suggestions, developers use their expertise to examine a small number of high-quality suggestions.

3 APPROACH

In this section we present the workflow that our novel approach and tool, MM-Assist, uses to automatically recommend and perform correctly the MoveMethod refactoring.

Figure 2 shows the architecture and the steps performed by MM-Assist. First, MM-Assist applies a set of pre-conditions that filter out the methods that cannot be safely moved, such as constructors (① in Figure 2). It then leverages vector embeddings from Language Models to identify methods

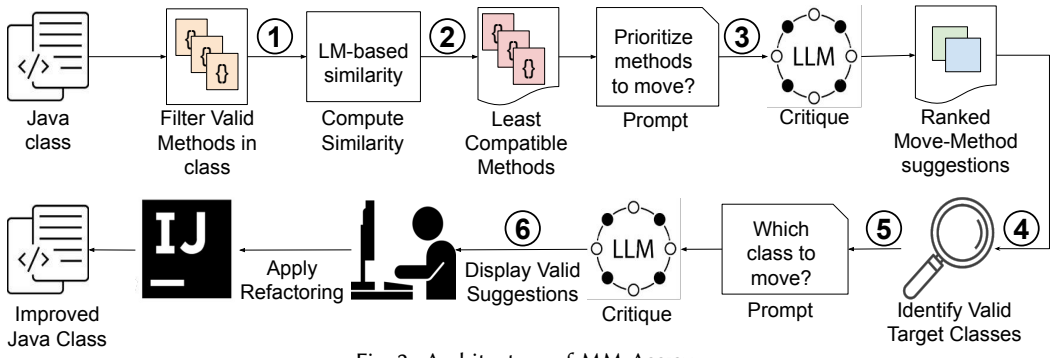

Fig. 2. Architecture of MM-Assist.

that are the least cohesive with their host class (② in Figure 2). In Figure 1, by comparing the embedding of `resolvePolicy` against the embedding of `EsqlSession` using cosine similarity, MM-Assist detects that this method might be misplaced (§ 3.2). It identifies other methods in `EsqlSession` that might be misplaced based on their low semantic relevance with the host class. Then, it passes the remaining candidates to the LLM (i.e., the method signature and the class body), which analyzes their implementation, dependencies, and relationships with the host class to prioritize the most promising MoveMethod recommendations (③ in Figure 2).

Once MM-Assist identifies candidate methods, it systematically evaluates potential target classes from the project codebase, considering method accessibility and static/non-static constraints (§ 3.3). For the `resolvePolicy` method, which utilizes the `enrichPolicyResolver` field (① in Figure 1), MM-Assist initially identifies several candidate classes, including `EnrichPolicyResolver` and `PolicyManager`.

To narrow down these candidates, MM-Assist calculates relevance scores between the candidate method and each potential target class. For `resolvePolicy`, these scores incorporate multiple factors: semantic coherence with `EnrichPolicyResolver`, structural relationships through the `enrichPolicyResolver` field reference, parameter compatibility (`policyNames`, `resolution`), and the overall functional context. MM-Assist subsequently ranks candidate classes based on these computed scores to identify the most viable targets (④ in Figure 2).

Using the Retrieval Augmented Generation (RAG) mechanism, we enhance the LLM's input with a prioritized list of target classes and their similarity metrics (⑤ in Figure 2). The LLM processes this enriched context to identify the optimal target class, factoring in code structure, dependencies, and semantic relevance. In this case, it correctly selects `EnrichPolicyResolver` as the appropriate destination for `resolvePolicy`, aligning with the developers' actual refactoring decision (see ③ in Figure 1). This combination of semantic relevance and LLM-based reasoning allows MM-Assist to produce actionable refactoring suggestions (⑥ in Figure 2). Finally, MM-Assist leverages the IDE's refactoring APIs to safely execute the recommended transformation automatically. Next, we discuss each of these steps and concepts in detail.

3.1 Important Concepts

Definition 3.1. (MoveMethod Refactoring) *A MoveMethod refactoring moves a method from a host class (where it currently resides but it doesn't belong) to a target class. We define a MoveMethod refactoring "ω" as a triplet* $(\mathtt{m}, \mathtt{H}, \mathtt{T})$*, which represents moving a method* `m` *from host class* `H` *to target class* `T`*. We denote a single MoveMethod refactoring with the symbol ω, and a set of MoveMethod refactorings with the symbol Ω. Further, for a given MoveMethod refactoring ω, we use the notation*

ω_m to denote the method to move ($\mathtt{m}$), ω_{H} to denote the host class (H) and ω_{T} to denote the target class (T) of the move.

Definition 3.2. (MoveMethod Recommendations) *We consider a list of MoveMethod refactoring candidates/suggestions, ordered by priority (most important at the beginning) to be MoveMethod Recommendations made by a tool, and denote this with the symbol $\mathfrak{R}$.*

We define $\mathfrak{R}$ to be on ordered list of MoveMethod suggestions.

This allow us to describe the top-N recommendations made by a system as the first N elements in $\mathfrak{R}$.

Definition 3.3. (Valid Refactoring Suggestion) We define valid refactoring suggestions as those that do not break the code. They are mechanically feasible: syntactically correct and successfully pass the pre-conditions as checked by the IDE. We differentiate between the validity of moving an instance and a static methods as follows:

(1) An **Instance Method** can be moved to a type in the host class' fields, or a type among the method's parameters. Several pre-condition checks are necessary to ensure the validity of the MoveMethod suggestion, including:
   - Method Movability - Is the method a part of the class hierarchy?
   - Access to references - Does the moved method loses access to the references it needs to perform its computation?

(2) A **Static Method** can be moved to almost any class in the project. A valid static-method move is one where the method can still access its references (e.g., fields, methods calls) from the new location.

We term any MoveMethod suggestion which is not valid to be an invalid suggestion.

Definition 3.4. (Invalid MoveMethod Recommendations) LLMs can generate many invalid MoveMethod refactorings that, if executed, would break a software system and produce compile errors. We classify these errors as hallucinations and categorize them as follows:

(1) **Target class do not exist (H1):** Formally, if $\mathbb{C}$ is the set of classes in a software system, then for a given MoveMethod suggestion ω, if $\omega_{\mathsf{T}} \notin \mathbb{C}$, we call it a hallucination.
(2) **Mechanically infeasible to move (H2):** The target class exists, but a refactoring suggestion is invalid according to definitions in the previous subsection 3.3.
(3) **Invalid Methods (H3)**: Methods that are parts of the software-design, and moving them requires multiple other refactoring changes to accommodate the move. For example, moving getters and setters would also need to be accompanied by moving the appropriate field.

3.2 Identifying Which Method To Move

We utilize LLMs for MoveMethod refactoring, hypothesizing that their extensive pre-training on vast code repositories and inherent understanding of code naturalness make them well-suited for identifying out-of-place methods. However, directly using LLMs to identify potential methods that may be misplaced within a class is risky, as it results in many invalid MoveMethod recommendations, starting with method-identification, to finding a valid target class.

Filter Invalid Candidates via Sanity Checks. Following established refactoring practices [9, 29, 47], we implement an initial filtering step to identify valid move method candidates. This sanity check eliminates methods that are likely already in the correct class. First, MM-Assist filters out getter and setter methods, as they cannot be moved without also relocating the associated fields. Next, it excludes methods involved in inheritance chains that can be overridden in child classes, since moving these would require additional structural changes. It also removes test methods and those with irrelevant content, such as empty bodies or methods consisting solely of comments. This initial filtering ensures that only viable candidates proceed to the next stages.

Identify Least Compatible Methods via Embedding-Based Analysis. To further refine candidate methods, we use an embedding-based analysis. An embedding is a vector representation that captures the semantic features of an entity (methods and classes) based on their content and relationships. We leverage VoyageAI embeddings [53], specifically trained on code, as they more effectively capture the semantic relevance of programming constructs. We use VoyageAI [53] due to its state-of-the-art performance in code-related tasks. Vectors are generated for two inputs: one for the method body and another for the host class, excluding the method body. Excluding the method ensures that the class embedding remains unbiased by the method itself. We then calculate the cosine similarity between these vectors to assess how well each method semantically aligns with its host class. Based on our analysis of the method distribution across classes in our oracle dataset, we observed a heavy-tailed distribution where 90% of classes contained fewer than 15 methods. Thus, we select either all methods in the class or the 15 least cohesive methods (based on cosine similarity scores), whichever is smaller. This systematic narrowing of the search space ensures only the most promising candidates advance, laying the groundwork for effective LLM-based prioritization.

Prioritize Methods with LLM Guidance. We leverage LLMs to determine if the tokens of a method and its body integrate cohesively with the rest of the class. Drawing inspiration from the success of Chain-of-Thought (CoT) reasoning in software engineering tasks [26, 38], we adopt a CoT approach, prompting the LLM to perform structured analyses: evaluate each method's purpose, cohesion, and dependencies, summarize the host class's responsibilities, and assess overall alignment. This step enables the LLM to gain a holistic understanding of the codebase before recommending methods for relocation. Our prompt is available in our replication package [42].

3.3 Recommending Suitable Target Classes

After identifying potential methods for relocation, the subsequent task is to determine the most appropriate target classes for these methods. However, this presents a substantial challenge, requiring a comprehensive analysis of the entire codebase. LLMs struggle with such tasks due to their limited context windows. To address this, we employ Retrieval Augmented Generation (RAG). RAG is a systematic approach designed to retrieve and refine relevant contextual information, thereby augmenting the input to the LLM. In our methodology, the task is to efficiently retrieve and augment the model with the most relevant target classes, informed by a combination of structural filtering and semantic comparison. Structural filtering refers to evaluating characteristics such as package proximity and utility class identification, while semantic comparison involves assessing the meaning and relationships of code elements to determine the most suitable target classes. Since we designed the retrieval process to enhance refactoring, we refer to this as "refactoring-aware RAG." We explain the details of the RAG process in the sections below.

Validating Target Classes via Sanity Checks. Once target classes are collected, sanity checks filter out unsuitable candidates (i.e., interfaces and duplicate signatures) to ensure mechanical feasibility to perform MoveMethod, following established refactoring practices [9, 29].

Target Class Retrieval. The target class retrieval process gathers relevant context about a candidate method and its host class. It then identifies other potential classes within the project for method relocation. This involves evaluating the proximity of each class to the host class in the package structure, and deciding if a class is a utility class, given that utility classes are often the most suitable targets for method relocations due to their general-purpose nature. Let $\mathtt{m}$ be the candidate method for relocation, $\mathtt{H}$ be the host class of $\mathtt{m}$, C be the set of all classes in the project, and $p_c \subseteq C$ be the set of potential target classes. We formalize the retrieval and ranking as follows:

$$p_c = \text{Retrieve}(\mathtt{H}, C) \tag{1}$$

where Retrieve is a function that selects potential target classes from the whole project.

Then, for each class $t_c \in p_c$, we compute a ranking score $R(c, \texttt{m})$ that considers both the structural proximity and utility nature of the class:

$$R(t_c, \texttt{m}) = w_p \cdot \text{proximity}(t_c, \texttt{H}) + w_u \cdot \text{isUtility}(t_c) \tag{2}$$

where:

- w_p is the weight assigned to package proximity (i.e., $w_p = 2$)
- w_u is the weight assigned to utility classes (i.e., $w_u = 1$)
- $\text{proximity}(t_c, \texttt{H})$ evaluates the package proximity between class t_c and the host class of method `m`
- $\text{isUtility}(t_c)$ is a boolean function that returns 1 if t_c is a utility class, and 0 otherwise

The resulting $R(t_c, \texttt{m})$ captures the weighted relevance of each potential target class, considering both structural and functional aspects of the codebase.

Semantic Relevance-Based Target Class Ranking. While static analysis offers a foundational understanding of valid refactoring opportunities, it often yields an overly broad set of potential target classes, as it lacks the ability to capture deeper semantic relationships and contextual nuances. As a result, static analysis alone cannot effectively prioritize or eliminate classes that are only superficially related to the candidate method. To refine this selection, we incorporate semantic relevance analysis, which evaluates both the content and intent of the candidate method and target classes, aiming to identify deeper semantic connections that static analysis may overlook.

Our semantic relevance analysis involves two key steps. First, MM-Assist extracts the method body out of the host class. Second, we utilize VoyageAI's embedding technique to compute the cosine similarity between the method body and potential target classes. This helps us to effectively capture semantic relationships between the method and target classes. Formally, the semantic relevance between a method and a class is computed as follows:

$$rel(\texttt{m}, t_c) = cosine(embed(\texttt{m}), embed(t_c)) \tag{3}$$

where embed represents the VoyageAI embedding function. We sort the target classes by their rel scores in descending order and select the top-k candidates ($k = 10$ based on empirical analysis).

Ranking Target Classes Using LLM. While embeddings capture semantic relevance, they primarily provide vector-based distance metrics. Adding LLM-based analysis enables us to leverage textual features like method names and fields that suggest design intent. In the final phase, MM-Assist augments the LLM's input with this enriched context. To avoid context overflow, we create a concise representation of each target class, including its name, field declarations, and method signatures. The LLM then takes as input the method to be moved along with these summarized target class representations, returning a prioritized list of target classes.

We can formalize this LLM-based ranking process as follows:

$$R_{\text{LLM}}(\texttt{m}) = \text{LLM}\left(\texttt{m}, p_c^k\right) \tag{4}$$

where:

- $R_{\text{LLM}}(\texttt{m})$ is the final ranked list of target classes for method `m`
- LLM represents the language model's decision-making function
- `m` is the candidate method for relocation
- p_c^k is the set of top k potential target classes ranked by $\text{rel}(\texttt{m}, t_c)$

After the LLM-based ranking process, MM-Assist presents the recommended method-class pairs to developers through an interactive interface, accompanied by a rationale explaining each suggestion. Upon developer selection of a specific recommendation, MM-Assist encapsulates the

approved method-target class pair into a refactoring command object. MM-Assist then executes the command automatically through the IDE's refactoring APIs, ensuring safe code transformation.

4 EMPIRICAL EVALUATION

To evaluate the effectiveness and usefulness of MM-Assist, we designed a comprehensive, multi-method evaluation to corroborate, complement, and expand research findings. This includes a formative study, comparative study, replication of real-world refactorings, repository mining, user/case studies, and questionnaire surveys. These methods, combining qualitative and quantitative data, work together to address four research questions.

RQ1. How effective are LLMs at suggesting MoveMethod refactoring opportunities? This question assesses vanilla LLMs' ability to identify and suggest refactoring opportunities. We conduct a formative study to understand the strengths and limitations of using LLMs directly for refactoring, examining the diversity, feasibility, and correctness of their suggestions.
RQ2. How effective is our approach, MM-Assist, at suggesting MoveMethod refactoring opportunities? We evaluate the performance of MM-Assist against the state-of-the-art tools, feTruth (representative for DL approaches) and JMove (representative for static analysis approaches). We use both a synthetic corpus used previously by other researchers and a new dataset of real refactorings performed by open-source developers.
RQ3. What is MM-Assist's runtime performance? This helps us understand MM-Assist's scalability and suitability for integration into developers' workflows. We assess its computational efficiency, measuring its runtime performance across various project sizes and complexity levels.
RQ4. How useful is our approach for developers? We focus on the utility of MM-Assist from a developer's perspective. We conduct a user study with 30 participants with industry experience who used MM-Assist on their authored code for one week. We analyze their ratings of the quality of recommendations, MM-Assist's usability, and its potential impact on refactoring practices.

4.1 Subject Systems

To evaluate LLMs' capability when suggesting MoveMethod refactoring opportunities, we employed two distinct datasets: a synthetic corpus widely used by previous researchers [9, 11, 25, 47] and a new corpus that contains real-world refactorings that open-source developers performed. Each corpus comes along with a "gold set" G of MoveMethod refactorings that a recommendation tool must attempt to match. We define G as a set of MoveMethod refactorings (see Definition 3.1) - each containing a triplet of method-to-move, host class, and target class $(\texttt{m}, \texttt{H}, \texttt{T})$.

Synthetic corpus. The *synthetic corpus* [47] was created by Terra et al. moving different methods `m` out of their original class `C` to a random destination class `C'`. The researchers then created the gold set as tuples (m, C', C), i.e., methods `m` that a tool should now move from `C'` back to its original class `C`. The researchers explained this ensures that method `m` is out of place in `C'` and can be moved back to original class `C`. This synthetic dataset moves *only instance methods*; it does not move *static methods*. This corpus consists of 10 open-source projects (i.e., Ant, Derby, DrJava, JfreeChart, JGroups, JTopen, JUnit, MvnForum, Lucene, Tapestry). On average, one project has 1,574 classes and 13,759 methods, with 207,163 LOC (the versions we used are on Replication Package [42]).

Real-world corpus. To complement the synthetic dataset and provide insights into how closely LLMs resemble the rationale of expert developers in real-world situations, we curated a corpus of *actual* MoveMethod refactorings that open-source developers performed on their projects. We construct this oracle using RefactoringMiner [51], the state-of-the-art tool for detecting refactorings.

We took extra precautions to prevent *LLM data contamination*, ensuring that the LLMs used by MM-Assist had no prior exposure to the data we tested and could not rely on previously memorized

results. With GPT-4's knowledge cutoff in October 2023, we focused our analysis on repository commits from January 2024 onward. Our initial dataset comprised the 25 most actively maintained Java repositories on GitHub, ranked by commit frequency and with over 1,000 stars.

For each commit where RefactoringMiner detected a MoveMethod refactoring, it reports the source and target classes, the original method's signature, and the moved method's signature. Many of these reported MoveMethod refactorings are false positives, often resulting from residual effects of other refactorings such as MoveClass (where an entire class is relocated to another package) or Extract Class (where a class is split into two, creating a new class along with the original). To filter out these false positives, we employed several techniques: first, we verified that both the source and target classes existed in both versions of the code (i.e., at the commit head and its previous head). For instance methods, we then checked if the method was moved to a field in the source class, to a parameter type, or if the moved method's parameters contained a reference to the source class (all preconditions for a valid MoveMethod). Starting from the instances detected by RefactoringMiner, we curated a dataset of 210 verified MoveMethod, with 102 static methods and 108 instance methods – on 12 projects (i.e., Elasticsearch, Spring-framework, Selenium, Ghidra, Vue-pro, Halo, Kafka, Graal, Redisson, selenium, Apache Flink, Spring-boot). On average, each project contains 8743 classes and 66306 methods spanning 1032344 LOCs. This oracle enables an evaluation of MM-Assist's performance on authentic refactorings made by experienced developers.

4.2 Effectiveness of LLMs (RQ1)

Evaluation Metrics. Using these datasets, we evaluated the recommendations made by the vanilla LLM and identified how many were hallucinations, as defined in Definition 3.4. We categorized hallucinations into three types: (i) recommendations where the target class does not exist in the project, (ii) recommendations where moving the method to the target class is mechanically infeasible due to the lack of a valid reference at the method's call sites, and (iii) recommendations that fail to meet the move method preconditions in Section 3.2.

Experimental Setup. We use the vanilla implementation of GPT-4o, a state-of-the-art LLM from OpenAI [7]. While MM-Assist is model agnostic (i.e., we can simply swap different models), we chose GPT-4o because other researchers [14, 41] show that it outperforms other LLMs when used for refactoring tasks. GPT-4o is also widely adopted in developer productivity tools [12, 21]. Our experimental setup was designed to assess the model's inherent capabilities in understanding and recommending MoveMethod without additional context or task-specific modifications. We formulated a prompt where we provided the source code in a given Host class and asked the LLM which methods are more appropriately placed in other classes and in which classes to move such methods. The exact prompt is in our companion webpage [42]. We set the temperature parameter of the LLM to 0 to obtain deterministic results.

For each host class in our Gold sets (both synthetic and real-world), we submitted the corresponding prompt to the LLM and collected its recommendations. We then evaluated them against the ground truth. Moreover, we conducted a qualitative analysis of the LLM's explanations and target class suggestions to gauge the depth of its understanding and possible hallucinations.

Table 1. Different kinds of hallucinations made by the Vanilla LLM

Corpus	# Recomms.	# Hall-class (H1)	# Hall-Mech (H2)	# Invalid Method (H3)
Synthetic (235)	723	362	168	51
Real-world (210)	1293	431	275	320

Results. Table 1 illustrates the distribution of valid suggestions and different types of hallucinations produced by the vanilla LLM for both synthetic and real-world datasets. We observed three

main types of hallucinations: Non-existent target classes (H1), where the LLM suggested moving methods to classes that don't exist; Unfeasible target classes (H2), where the proposed refactorings would break compilation due to inaccessible target classes; and Incorrect method identification (H3), where the LLM mistakenly flagged well-placed methods for relocation. *Crucially, actuating any of these hallucinations would lead to broken code, compilation errors, or degraded software design.*

In the synthetic dataset, comprising 723 total suggestions, a mere 20% (142) were valid. The overwhelming 80% (581) were hallucinations, with H1 accounting for 50.1% (362), H2 for 23.2% (168), and H3 for 7.1% (51) of all suggestions. The real-world dataset also presented significant challenges. Once again, out of 1293 total suggestions, only 20% were valid. The 80% hallucinations were distributed as follows: H1 comprised 33% (431), H2 22% (275), and H3 25% (320) of all suggestions.

These findings underscore the impracticality of using vanilla LLM suggestions for MoveMethod without extensive filtering and validation. For every valid suggestion, a developer would need to sift through and discard 3-4 invalid ones, many of which could introduce critical errors if implemented. This not only undermines the potential time-saving benefits of automated refactoring but also introduces significant risks of introducing bugs or degrading code quality.

LLMs excel at generating MoveMethod recommendations, yet only 20% of these suggestions are useful.

4.3 Effectiveness of MM-Assist (RQ2)

To evaluate MM-Assist's effectiveness, we conducted a comparative study against the state-of-the-art MoveMethod recommendation tools. We directly compare with the best-in-class tools: JMove [47] for static analysis and feTruth [29] for ML/DL approaches, as both have been shown to outperform all previous tools. We also compare with the Vanilla-LLM (GPT 4o), which represents the standard LLM solution (without using MM-Assist's enhancements). We went the extra mile to ensure a fair comparison: we consulted with the feTruth and JMove authors to ensure the optimal tool's settings, and clarified with the authors when their tools did not produce the expected results.

Evaluation Metrics. For evaluation, we employ recall-based metrics, following an approach similar to that used in the evaluation of JMove [47], a well-established tool in this domain. For refactoring recommendation solutions that aim to be used by industry practitioners, recall@k (where k is a small number so that practitioners do not sift through several candidates) is a more appropriate metric. Moreover, it is more fair than precision as it is *not subjective*. Given that a tool can produce reasonable recommendations, but only the authors of those software systems can accurately determine whether those recommendations are (un)reasonable, we cannot calculate a true precision. But we can calculate a true recall as we have a reliable Gold set (G).

We present recall for each phase of suggesting the move-method refactoring: first, identifying that a method is misplaced ($Recall_M$); second, identifying a target class for the misplaced method ($Recall_C$); third, identifying the entire chain of refactoring: selecting the right method and the right target class ($Recall_{MC}$).

For a recommendation list $\mathfrak{R} = (\Omega, \tau)$, we define $Recall_M$, $Recall_C$ and $Recall_{MC}$ as follows:

$Recall_M = \frac{|\Omega_m|}{|G|}$, $Recall_C = \frac{|\Omega_m \cap G|}{|\Omega_m|}$, and $Recall_{MC} = \frac{|(\Omega \cap G)|}{|G|}$

, Where Ω_m is the subset of Ω containing refactorings whose method components match those in the ground truth set G. Formally, we define it as follows:

$\Omega_m = \{\omega | \omega \in \Omega \wedge \exists(\omega_m, \omega_c, *) \in G\}$

For each recall metric, we calculate Recall@k for the top k recommendations, where k $\in \{1, 2, 3\}$.

Experimental Setup. Using the gold set, we trigger each tool on a source class for which one of its methods was moved. We applied each tool to both the synthetic corpus and the real-world

Table 2. Recall rates of MM-Assist on the synthetic corpus of 235 refactorings [47] that moved instance methods. $Recall_M$ = identify the method, $Recall_C$ = identify the target class for a given method, $Recall_{MC}$= identify the method&target class pair

Approach	$Recall_M$			$Recall_C$			$Recall_{MC}$		
	@1	@2	@3	@1	@2	@3	@1	@2	@3
JMove	41.3%	43.0%	43.4%	97.1%	97.1%	97.1%	40.0%	41.7%	42.1%
feTruth	2.1%	2.9%	3.4%	**100%**	**100%**	**100%**	2.1%	2.9%	3.4%
Vanilla-LLM	73.1%	77.3%	80.8%	72.8%	72.8%	72.8%	55.6%	58.2%	60.2%
MM-Assist	**75.7%**	**81.2%**	**82.9%**	96.5%	99.5%	99.5%	**73.2%**	**78.7%**	**80.4%**

dataset, recording the suggestions generated for each input. Considering the number of entries in the datasets, given that JMove can take a long time to run (12+ hours on a large project), we cutoff its execution after 1 hour. The tools generate a ranked list of possible target classes (T) for each candidate method (m). We then compared these suggestions against the ground truth to calculate recall for each tool using the evaluation metrics presented earlier. We computed $Recall_M$, $Recall_C$, and $Recall_{MC}$ based on each tool's recommendations. If a tool correctly identifies the target method as a candidate for a move (no matter the recommended target class), we count it in $Recall_M$. Starting from correctly identified target methods, we calculated the number of cases where a tool found the correct target class, thus computing $Recall_C$. Finally, we counted cases where the exact MoveMethod refactoring was recommended, thus computing $Recall_{MC}$.

Results. Table 2 shows the effectiveness of MM-Assist and baseline tools on the synthetic dataset. As shown, MM-Assist demonstrates superior performance across many of the recall metrics. Most importantly, it shows an increase of 17-71% compared to all baselines in $Recall_{MC}$@1, which is the most comprehensive measure as it captures both the correct identification of misplaced methods and the accurate suggestion of target classes. While JMove exhibits high accuracy in target class identification ($Recall_C$@1 = 97.1%), it shows limitations in method identification. Interestingly, feTruth achieves perfect $Recall_C$ but extremely low $Recall_M$ (2% – 3%) despite being very prolific in recommending as many as 67 methods to be moved from a class (see Replication Package [42]). When it does correctly identify a method, it accurately suggests the target class. However, this high $Recall_C$ is less meaningful, thus resulting in a low overall $Recall_{MC}$ due to the low number of correctly identified misplaced methods. We confirmed this paradox with feTruth authors.

The Vanilla-LLM shows promise in method identification but fails to suggest the target class accurately. In contrast to all previous tools, MM-Assist's consistent high performance across all metrics, especially in ($Recall_{MC}$@3 = 80.4%), highlights the effectiveness of our approach in combining LLM capabilities with static analysis and semantic relevance.

With the 2024 real-world dataset of refactorings performed by open-source developers, we found a wider difference in performance based on changing two variables. First, we distinguish between cases when the MoveMethod target was an instance or static method – thus we shed light on the effectiveness of MM-Assist when using different mechanisms to suggest a target class. Second, we distinguish between small and large classes based on their method count. Our analysis using the real-world oracle reveals a heavy-tail distribution, where we label classes with fewer than 15 methods (90th percentile across all projects) as Small Classes and the rest as Large Classes.

Tables 3 and 4 summarize our results for instance and static method moves, respectively. Since JMove could not finish running on the entire dataset, we note the number of completed entries in parenthesis. For instance methods, MM-Assist achieved 8%-80% higher recall compared to baseline tools when handling smaller classes. However, we also observed a performance decrease in all

Table 3. Recall rates on 108 instance methods moved by OSS developers in 2024. First column shows the number of small or large classes in the oracle. $Recall_M$ = identify the method, $Recall_C$ = identify the target class for a previously identified method to be moved, $Recall_{MC}$= identify the method&target class pair.

Oracle Size	Approach	$Recall_M$			$Recall_C$			$Recall_{MC}$		
		@1	@2	@3	@1	@2	@3	@1	@2	@3
SmallClasses (38)	JMove (19)	5%	5%	5%	0%	0%	0%	0%	0%	0%
	feTruth	20%	20%	20%	**100%**	**100%**	**100%**	20%	20%	20%
	Vanilla-LLM	55%	68	73%	86%	86%	86%	63%	58%	63%
	MM-Assist	**76%**	**92%**	**94%**	86%	89%	89%	**71%**	**82%**	**82%**
LargeClasses (70)	JMove (24)	8%	8%	8%	**100%**	**100%**	**100%**	8%	8%	8%
	feTruth	2%	8%	12%	76%	76%	76%	2%	6%	9%
	Vanilla-LLM	7%	12%	14%	76%	76%	76%	10%	7%	10%
	MM-Assist	**37%**	**41%**	**51%**	75%	79%	79%	**32%**	**35%**	**40%**

Table 4. Recall rates on 102 static methods moved by OSS developers in 2024. $Recall_M$ = identify the method, $Recall_C$ = identify the target class for a given method, $Recall_{MC}$= identify the method&target class pair.

Oracle Size	Approach	$Recall_M$			$Recall_C$			$Recall_{MC}$		
		@1	@2	@3	@1	@2	@3	@1	@2	@3
SmallClasses (40)	feTruth	7%	15%	15%	14%	14%	14%	1%	2%	2%
	Vanilla-LLM	43%	57%	65%	7%	7%	7%	3%	4%	5%
	MM-Assist	**55%**	**65%**	**70%**	**21%**	**21%**	**21%**	**12%**	**14%**	**15%**
LargeClasses (62)	feTruth	6%	11%	15%	6%	6%	6%	0.4%	0.6%	0.8%
	Vanilla-LLM	**14%**	25%	**34%**	9%	9%	9%	1.3%	2.2%	3.0%
	MM-Assist	**14%**	**27%**	29%	**44%**	**44%**	**44%**	**6%**	**12%**	**13%**

tools when identifying MoveMethod opportunities in large classes. This is because it is more likely for large classes to suffer from significant technical debt, and there are many candidate methods that can be moved out of it – thus it is harder to pick the proverbial "needle from the haystack". Since we are not the developers of these classes, we are not the best to judge the merit of each recommendation, thus we rely on whether the tool matched exactly what the open-source developers refactored. While recalling instance MoveMethod, we noticed that our performance on small classes was comparable to the synthetic dataset, while for other tools dropped significantly. Notably, our $Recall_C$@3 was 89%, which can be attributed to the performance of the LLM in picking the suitable target classes.

However, the differences are more nuanced when we evaluate MM-Assist on static methods: we find that our $Recall_C$ drops significantly. This is because the scope of moving static methods is massive - they can be moved to (almost) any class in the project. For large projects like Elasticsearch, this means picking the right target class among 21615 candidates. The real-world oracle contains on average 8743 classes per project. This shows that recommending which static methods to move is a much harder problem than recommending instance methods, as analyzing thousands of classes to find the right one is hard. This could explain why prior MoveMethod tools do not give recommendations for moving static methods. As we are the first ones to make strides in this harder problem, we hope that by contributing this dataset of static MoveMethod to the research community, we stimulate growth in this area.

4.4 Runtime performance of MM-Assist (RQ3)

Experimental Setup. We used both the synthetic and real-world corpus employed in our earlier experiments to measure the total execution time. The execution time is an interval between the tool's triggering on a source class and the display of final refactoring suggestions to the user. To understand what components of MM-Assist take the most time, we also measured the amount of time it took to generate responses from the LLM, and the time it took to process suggestions. To ensure real-world applicability, we conducted these measurements using the MM-Assist plugin for IntelliJ IDEA, mirroring the actual usage scenario for developers. This approach allows us to account for any overhead introduced by the IDE integration, providing a more accurate representation of the tool's performance in practical settings. We conducted all experiments on a commodity laptop, an M1 MacBook Air with 16GB of RAM.

Results. Our empirical evaluation demonstrates that MM-Assist achieves an average runtime of 27.5 seconds for generating suggestions. The primary computational overhead stems from the LLM API interactions consuming approximately 9 seconds. In our experience with JMove, on the larger projects in our real-world dataset, JMove takes several hours (up to 24 hours) to complete running – thus we imposed the 1-hour cutoff time. Out of the box, feTruth is also slow and can take 12+ hours to run on large project. With the help of feTruth authors, we were able to run it on a single class at a time – this takes an average of 6 minutes per class. Thus, compared with the baseline approaches, MM-Assist is two orders of magnitude and one order of magnitude faster than JMove and feTruth, respectively.

4.5 Usefulness of MM-Assist (RQ4)

We designed a user study to assess the practical utility of MM-Assist from a developer's perspective.

Dataset. We made the deliberate choice to have participants use projects with which they were familiar. This decision was grounded in several key considerations. First, by working with familiar codebases, developers can leverage their deep understanding of the project's architecture, design decisions, and evolution history. This familiarity enables them to make more informed judgments about the appropriateness and potential impact of the suggested refactorings. Second, using personal projects enhances the validity of our study, as it closely mimics real-world scenarios where developers refactor code they have either authored or maintained extensively. Third, this approach allows us to capture a diverse range of project types, sizes, and domains, potentially uncovering insights that might be missed in a more constrained, standardized dataset.

Experimental Setup. 30 students (25 Master's and 5 Ph.D. students) volunteered to participate in our study. Based on demographic information provided by the participants, 73% have industrial experience. All participants, with the exception of two, have experience with the Java programming language. Finally, the majority of participants (24 out of 30) have prior experience with refactoring.

We instructed the participants to use MM-Assist for a week and run it on at least ten different Java classes from projects they work on. The selection of these classes was left to the discretion of the participants, with the guidance to choose files they had either authored or were very familiar with. For each class they selected, MM-Assist provided up to three MoveMethod recommendations. We chose to present three recommendations to strike a balance between variety and practicality. Afterward, they sent us the fine-grained telemetry data from the plugin usage. For confidentiality reasons, we anonymize the data by stripping away any sensitive information about their code, e.g., the names and source code of classes or methods that MM-Assist presented to them. We collected usage statistics from each invocation of the plugin on each class. In particular, we collected this information: how the users rated each individual recommendation and whether they finally changed their code based on the recommendation.

Participants rated each recommendation on a 6-point Likert scaled ranging from (1) Very unhelpful to (6) Very helpful. We chose this 6-point Likert scale to force a non-neutral stance, encouraging participants to lean towards either a positive or negative assessment of each recommendation. We asked the participants to rate the MM-Assist's recommendations while they were fresh in their minds, right after they analyzed each recommendation.

After participants sent their usage telemetry, we asked them to fill out an anonymous survey asking about their experience using MM-Assist. We asks participants to compare MM-Assist's workflow against the IDE, and asked for open-ended feedback about their experience.

Results. 30 participants applied MM-Assist on 350 classes. We found that, in 290 classes (out of 350 classes) the participant positively rated one of the recommendations (82.8% of the time). Additionally, we found that the users accepted and applied a total of 216 refactoring recommendations on their code, i.e., 7 refactorings per user, on average. This shows that MM-Assist is effective at generating useful recommendations that developers, who are familiar with their code, accept.

The participants also provided feedback in free-form text. Of the 30 participants, 80% of them rated the plugin's experience highly, when comparing against the workflow in the IDE. In praise of MM-Assist, the participants said that MM-Assist gave them a sense of control, allowing them to apply refactorings that they agreed with. Others noted that plugin's presentation of suggestions was good. The participants also provided feedback about ways to improve MM-Assist's UI experience, for example some said that the rationale that LLM gave for the recommendations could be improved.

5 DISCUSSION

Internal Validity: Dataset bias poses a potential threat to the effectiveness of MM-Assist. To mitigate this, we employ both a synthetic dataset (widely used by others), offering a controlled environment, *and* a real-world dataset comprising refactorings performed by open-source developers.

External Validity: This concerns the generalizability of our results. Because we rely on a specific LLM, it may impact the broader applicability of our findings. We anticipate that advancements in LLM technology will improve overall performance, though this needs to be verified empirically. Second, MM-Assist currently focuses on Java code. Although our approach is conceptually language-agnostic, with prompts not tied to Java-specific constructs, we cannot definitively claim generalizability to other programming languages without further investigation. Future work will explore the effectiveness of MM-Assist across diverse programming languages.

Tool implementation. MM-Assist's architecture supports extensibility across programming languages and development environments through three fundamental design decisions. First, our core refactoring logic separates language-specific concerns from the refactoring workflow. While the current implementation uses IntelliJ's refactoring framework for Java, future versions could integrate with the Language Server Protocol (LSP) [2]. LSP is a standardized protocol that enables development tools to provide consistent programming language support across different editors and IDEs. Second, we designed a modular pipeline where components communicate through well-defined interfaces. The LLM service, embedding model, and IDE integration are independent modules that can be replaced without affecting the rest of the system. For instance, developers can swap different embedding models while retaining the same similarity computation logic or integrate different IDEs through their native plugin APIs. While our evaluation shows that general-purpose LLMs like GPT-4o, which also perform well at code understanding [1], our modular architecture allows easy integration of code-specific LLMs (e.g., StarCoder [27], CodeLlama [43]) through the same interface. This opens interesting research directions to evaluate whether models specifically trained on code repositories could offer better refactoring suggestions. To address the non-deterministic nature of LLMs, we set the temperature parameter to 0 in all LLM calls,

which makes the model's output deterministic for identical inputs. For IDE developers, MM-Assist demonstrates how to safely integrate AI-powered suggestions with existing IDE infrastructure.

6 RELATED WORK

We organize the related work into two parts: (i) the literature on MoveMethod refactoring, and (ii) the research that use LLMs for software refactoring.

Move method refactoring. The refactoring process, as found in the survey conducted by Mens and Tourwe [34], is a pipeline of steps that encompasses identifying areas for refactoring, selecting the appropriate refactoring type, ensuring existing software behavior is preserved, implementing changes, evaluating their impact on quality and process metrics, and maintaining consistency with related documentation and artifacts. JMove [47] and JDeodorant [48] detect methods located in incorrect classes and suggest moving them to more appropriate classes based on software metrics derived from static analysis. HMove [9], a recently introduced tool, leverages a graph representation of code entities and LLMs, relying on several user-configured thresholds to detect and recommend move method refactorings. Additionally, Sales et al. [44] recommends move method refactorings based on the set of static dependencies established by a particular method, while Bavota et al. [5] uses relational topic models to identify methods to move. Liu et al. [31] further expands on move method refactoring by identifying other methods that should be moved once a method is relocated by a developer. In contrast to our approach, a major difference with these tools is that they require users to define thresholds, which can be challenging for inexperienced users. On the other hand, RMove [11] utilizes both structural and semantic representations from code snippets and employs machine learning models to recommend move method refactoring opportunities.

Most of these techniques recommend or partially automate refactoring activities but do not fully automate the whole refactoring pipeline [34], unlike our approach. Moreover, MM-Assist attacks the problem drastically differently. Previous tools compute whole project dependencies (which is computationally expensive and doesn't scale) and then produce a confidence score for each method in the project. Thus, they treat this as a *classification* instead of a *recommendation* problem: they produce dozens of possible prospects to move out a given class (many of which are unuseful), which burdens the programmer. In contrast, MM-Assist provides up to 3 reasonable recommendations per class, and most of them align with how expert developers refactor the code.

Refactoring in the age of LLMs. A recent systematic study [22] analyzing 395 research papers demonstrates that LLMs are being employed to solve various software engineering tasks. While code generation has been the predominant application, recently LLMs like ChatGPT have been applied to automate code refactoring [3, 9, 13, 46]. Cui et al. [10] leverage intra-class dependency hypergraphs with LLMs to perform extract class refactoring, while iSMELL [55] uses LLMs to detect code smells and suggest corresponding refactorings. However, LLMs are prone to hallucinations, which can introduce incorrect or broken code, posing challenges for automated refactoring systems. Unlike other approaches, our method addresses this limitation by validating and ranking LLM-generated outputs, ensuring that developers can safely execute refactoring recommendations.

The prevalence of hallucinations in LLM-based refactoring is widely studied. Pomian et al. [40, 41] investigated hallucinations in ExtractMethod refactoring, while Dilhara et al. [14] analyzed hallucinations in Python code modifications. These studies consistently show that LLMs can hallucinate during refactoring tasks, substantiating our findings, where LLMs hallucinated in 80% of the cases when suggesting MoveMethod. This highlights the necessity of robust validation mechanisms, which are integral to our tool, MM-Assist, ensuring the reliability and safety of refactoring suggestions generated by LLMs.

7 CONCLUSIONS

Despite lots of active research in the area of recommending MoveMethod refactorings, the progress over the years has been incremental and has stifled. The rise of LLMs and their applications to the field of refactoring has rejuvenated the field. Our approach and tool, MM-Assist, significantly outperforms previous best-in-class tools and provides recommendations that better align with the practices of expert developers. When replicating a corpus of refactorings performed in 2024 by open source developers, MM-Assist improves the recall over previous best-in-class tools by 4x, while running in 10x–100x less time. Moreover, our case study with 30 experienced participants who used MM-Assist to refactor their own code for one week shows they rated 82.8% of MM-Assist recommendations positively.

The key to unleashing these breakthroughs is combining static and semantic analysis to (i) eliminate LLM hallucinations and (ii) focus its laser. MM-Assist checks refactoring preconditions automatically which cuts down the LLM hallucinations. Moreover, by leveraging semantic embedding into a RAG approach, MM-Assist narrows down the context for the LLM so that it can focus its laser on a small number of high-quality prospects. This was instrumental in picking the right candidate from a population of more than 21000 candidate target classes. We hope that these techniques inspire others with fresh ideas on how to solve many other refactoring recommendation domains such as splitting large classes or packages.

REFERENCES

[1] 2024. Chatbot Arena Leaderboard. https://huggingface.co/spaces/lmsys/chatbot-arena-leaderboard.

[2] 2024. Language Server Protocol (LSP). https://github.com/python-rope/pylsp-rope.

[3] Eman Abdullah AlOmar, Anushkrishna Venkatakrishnan, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. 2024. How to refactor this code? An exploratory study on developer-ChatGPT refactoring conversations. In *Proceedings of the 21st International Conference on Mining Software Repositories* (Lisbon, Portugal) *(MSR '24)*. Association for Computing Machinery, New York, NY, USA, 202–206. https://doi.org/10.1145/3643991.3645081

[4] Gabriele Bavota, Andrea De Lucia, and Rocco Oliveto. 2011. Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software* 84, 3 (2011), 397–414. https://doi.org/10.1016/j.jss.2010.11.918

[5] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2014. Methodbook: Recommending Move Method Refactorings via Relational Topic Models. *IEEE Transactions on Software Engineering* 40, 7 (2014), 671–694. https://doi.org/10.1109/TSE.2013.60

[6] Timofey Bryksin, Evgenii Novozhilov, and Aleksei Shpilman. 2018. Automatic recommendation of move method refactorings using clustering ensembles. In *Proceedings of the 2nd International Workshop on Refactoring* (Montpellier, France) *(IWoR 2018)*. Association for Computing Machinery, New York, NY, USA, 42–45. https://doi.org/10.1145/3242163.3242171

[7] ChatGPT [n. d.]. OpenAI. https://openai.com/.

[8] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E. Gonzalez, and Ion Stoica. 2024. Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference. arXiv:2403.04132 [cs.AI]

[9] Di Cui, Jiaqi Wang, Qiangqiang Wang, Peng Ji, Minglang Qiao, Yutong Zhao, Jingzhao Hu, Luqiao Wang, and Qingshan Li. 2024. Three Heads Are Better Than One: Suggesting Move Method Refactoring Opportunities with Inter-class Code Entity Dependency Enhanced Hybrid Hypergraph Neural Network. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 745–757.

[10] Di Cui, Qiangqiang Wang, Yutong Zhao, Jiaqi Wang, Minjie Wei, Jingzhao Hu, Luqiao Wang, and Qingshan Li. 2024. One-to-One or One-to-Many? Suggesting Extract Class Refactoring Opportunities with Intra-class Dependency Hypergraph Neural Network. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1529–1540.

[11] Di Cui, Siqi Wang, Yong Luo, Xingyu Li, Jie Dai, Lu Wang, and Qingshan Li. 2022. RMove: Recommending Move Method Refactoring Opportunities using Structural and Semantic Representations of Code. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 281–292. https://doi.org/10.1109/ICSME55016.2022.00033

[12] Cursor. [n. d.]. Cursor. https://www.cursor.com/. Accessed: 2024-10-07.

[13] Kayla DePalma, Izabel Miminoshvili, Chiara Henselder, Kate Moss, and Eman Abdullah AlOmar. 2024. Exploring ChatGPT's code refactoring capabilities: An empirical study. *Expert Systems with Applications* 249 (2024), 123602.

[14] Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, and Danny Dig. 2024. Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example. 1, FSE, Article 29 (jul 2024), 23 pages. https://doi.org/10.1145/3643755

[15] Malinda Dilhara, Danny Dig, and Ameya Ketkar. 2023. PYEVOLVE: Automating Frequent Code Changes in Python ML Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 995–1007. https://doi.org/10.1109/ICSE48619.2023.00091

[16] 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.* (jul 2021). https://doi.org/10.1145/3453478

[17] 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 (ICSE '22)*. https://doi.org/10.1145/3510003.3510225

[18] Elasticsearch. [n. d.]. Move Method Refactoring in the Elasticsearch Project. https://github.com/elastic/elasticsearch/commit/876e70159c01ae306251281ae2fdbabca8732ed9.

[19] Sarah Fakhoury, Devjeet Roy, Adnan Hassan, and Vernera Arnaoudova. 2019. Improving source code readability: Theory and practice. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE.

[20] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.*

[21] Github. [n. d.]. Copilot. https://github.com/features/copilot. Accessed: 2024-10-07.

[22] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* (Sept. 2024). https://doi.org/10.1145/3695988 Just Accepted.

[23] James Ivers, Anwar Ghammam, Khouloud Gaaloul, Ipek Ozkaya, Marouane Kessentini, and Wajdi Aljedaani. 2024. Mind the Gap: The Disconnect Between Refactoring Criteria Used in Industry and Refactoring Recommendation Tools.

In *International Conference on Software Maintainance and Evolution*. IEEE.

[24] Zarina Kurbatova, Ivan Veselov, Yaroslav Golubev, and Timofey Bryksin. 2020. Recommendation of Move Method Refactoring Using Path-Based Representation of Code. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (Seoul, Republic of Korea) *(ICSEW'20)*. Association for Computing Machinery, New York, NY, USA, 315–322. https://doi.org/10.1145/3387940.3392191

[25] Zarina Kurbatova, Ivan Veselov, Yaroslav Golubev, and Timofey Bryksin. 2020. Recommendation of move method refactoring using path-based representation of code. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 315–322.

[26] Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. [n. d.]. CodeChain: Towards Modular Code Generation Through Chain of Self-revisions with Representative Sub-modules. In *The Twelfth International Conference on Learning Representations*.

[27] R Li, LB Allal, Y Zi, N Muennighoff, D Kocetkov, C Mou, M Marone, C Akiki, J Li, J Chim, et al. 2023. StarCoder: May the Source be With You! *Transactions on machine learning research* (2023).

[28] Yun Lin, Xin Peng, Yuanfang Cai, Danny Dig, Diwen Zheng, and Wenyun Zhao. 2016. Interactive and guided architectural refactoring with search-based recommendation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 535–546. https://doi.org/10.1145/2950290.2950317

[29] Bo Liu, Hui Liu, Guangjie Li, Nan Niu, Zimao Xu, Yifan Wang, Yunni Xia, Yuxia Zhang, and Yanjie Jiang. 2023. Deep Learning Based Feature Envy Detection Boosted by Real-World Examples. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 908–920. https://doi.org/10.1145/3611643.3616353

[30] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. 2021. Deep Learning Based Code Smell Detection. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1811–1837. https://doi.org/10.1109/TSE.2019.2936376

[31] Hui Liu, Yuting Wu, Wenmei Liu, Qiurong Liu, and Chao Li. 2016. Domino Effect: Move More Methods Once a Method is Moved. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 1–12. https://doi.org/10.1109/SANER.2016.14

[32] Hui Liu, Zhifeng Xu, and Yanzhen Zou. 2018. Deep learning based feature envy detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 385–396. https://doi.org/10.1145/3238147.3238166

[33] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics (TACL)* 0 (2023).

[34] T. Mens and T. Tourwe. 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004), 126–139. https://doi.org/10.1109/TSE.2004.1265817

[35] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* (2012). https://doi.org/10.1109/TSE.2011.41

[36] needleInHaystack [n. d.]. Needle In A Haystack - Pressure Testing LLMs. https://github.com/gkamradt/LLMTest_NeedleInAHaystack.

[37] 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.).

[38] Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. NExT: Teaching Large Language Models to Reason about Code Execution. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*, Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (Eds.). PMLR, 37929–37956. https://proceedings.mlr.press/v235/ni24a.html

[39] William F. Opdyke. 1992. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. Ph. D. Dissertation. University of Illinois at Urbana-Champaign.

[40] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Timofey Bryksin, and Danny Dig. 2024. Next-Generation Refactoring: Combining LLM Insights and IDE Capabilities for Extract Method. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

[41] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Andrey Sokolov, Timofey Bryksin, and Danny Dig. 2024. EM-Assist: Safe Automated ExtractMethod Refactoring with LLMs. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 582–586. https://doi.org/10.1145/3663529.3663803

[42] ReplicationPackage [n. d.]. MM-Assist replication package. https://mm-assist.netlify.app/.

[43] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[44] Vitor Sales, Ricardo Terra, Luis Fernando Miranda, and Marco Tulio Valente. 2013. Recommending Move Method refactorings using dependency sets. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. 232–241. https://doi.org/10.1109/WCRE.2013.6671298

[45] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. 2019. Automatically assessing code understandability. *IEEE Transactions on Software Engineering* (2019).

[46] Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanobe. 2023. Refactoring programs using large language models with few-shot examples. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 151–160.

[47] Ricardo Terra, Marco Tulio Valente, Sergio Miranda, and Vitor Sales. 2018. JMove: A novel heuristic and tool to detect move method refactoring opportunities. *Journal of Systems and Software* 138 (2018), 19–36. https://doi.org/10.1016/j.jss.2017.11.073

[48] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2018. Ten years of JDeodorant: Lessons learned from the hunt for smells. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 4–14. https://doi.org/10.1109/SANER.2018.8330192

[49] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of Move Method Refactoring Opportunities. *IEEE Transactions on Software Engineering* 35, 3 (May 2009), 347–367. https://doi.org/10.1109/TSE.2009.1

[50] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* (2022). https://doi.org/10.1109/TSE.2020.3007722

[51] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/3180155.3180206

[52] Szymon Tworkowski, Konrad Staniszewski, Mikoł aj Pacek, Yuhuai Wu, Henryk Michalewski, and Piotr Mił oś. 2023. Focused Transformer: Contrastive Training for Context Scaling. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 42661–42688. https://proceedings.neurips.cc/paper_files/paper/2023/file/8511d06d5590f4bda24d42087802cc81-Paper-Conference.pdf

[53] VoyageAI. [n. d.]. Voyage AI Embeddings. https://docs.voyageai.com/docs/embeddings.

[54] Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2024. CodeRAG-Bench: Can Retrieval Augment Code Generation? *arXiv preprint arXiv:2406.14497* (2024).

[55] Di Wu, Fangwen Mu, Lin Shi, Zhaoqiang Guo, Kui Liu, Weiguang Zhuang, Yuqi Zhong, and Li Zhang. 2024. iSMELL: Assembling LLMs with Expert Toolsets for Code Smell Detection and Refactoring. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1345–1357. https://doi.org/10.1145/3691620.3695508