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

Abhiram Bellur
University of Colorado
abhiram.bellur@colorado.edu

Fraol Batole
Tulane University
fbatole@tulane.edu

Mohammed Raihan Ullah
University of Colorado
raihan.ullah@colorado.edu

Malinda Dilhara
Amazon Web Services
malwala@amazon.com

Yaroslav Zharov
JetBrains Research
yaroslav.zharov@jetbrains.com

Timofey Bryksin
JetBrains Research
timofey.bryksin@jetbrains.com

Kai Ishikawa
NEC Corporation
k-ishikawa@nec.com

Haifeng Chen
NEC Laboratories America
haifeng@nec-labs.com

Masaharu Morimoto
NEC Corporation
m-morimoto@nec.com

Takeo Hosomi
NEC Corporation
takeo.hosomi@nec.com

Tien N. Nguyen
University of Texas at Dallas
tien.n.nguyen@utdallas.edu

Hridesh Rajan
Tulane University
hrajan@tulane.edu

Nikolaos Tsantalis
Concordia University
nikolaos.tsantalis@concordia.ca

Danny Dig
University of Colorado, JetBrains Research
danny.dig@colorado.edu

***Abstract*—MOVEMETHOD is a hallmark refactoring. Despite a plethora of research tools that recommend which methods to move and where, these recommendations do not align with how expert developers perform MOVEMETHOD. Given the extensive training of Large Language Models and their reliance upon *naturalness of code*, they should expertly recommend which methods are *misplaced* in a given class and which classes are better hosts. 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 fully powered assistant for MOVEMETHOD refactoring that automates its whole end-to-end lifecycle, from recommendation to execution. We designed novel solutions that 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. As MOVEMETHOD refactoring requires global, project-level reasoning, we solved the limited context size of LLMs by employing *refactoring-aware retrieval augment generation* (RAG). Our approach, MM-ASSIST, synergistically combines the strengths of the LLM, IDE, static analysis, and semantic relevance. In our thorough, multi-methodology empirical evaluation, we compare MM-ASSIST with the previous state-of-the-art approaches. MM-ASSIST significantly outperforms them: (i) on a benchmark widely used by other researchers, our Recall@1 and Recall@3 show a 1.7x improvement; (ii) on a corpus of 210 recent refactorings from Open-source software, our Recall rates improve by at least 2.4x. Lastly, we conducted a user 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.

I. INTRODUCTION

MOVEMETHOD is a key refactoring [1] that relocates a misplaced method to a more suitable class. A method is *misplaced* if it interacts more with another class' state than its own. MOVEMETHOD improves modularity by aligning methods with relevant data, enhances cohesion, and reduces coupling. It removes code smells like FeatureEnvy [2], GodClass [3], DuplicatedCode [4], and MessageChain [1], reducing technical debt. It ranks among the top-5 most common refactorings [5–7], both manually and automatically performed.

The MOVEMETHOD lifecycle has four phases: (i) identifying a misplaced method `m` in its host class `H`, (ii) finding a suitable target class `T`, (iii) ensuring refactoring pre- and post-conditions to preserve behavior, and (iv) executing the transformation. Each phase is challenging—identifying candidates requires understanding design principles and the codebase, while checking preconditions [2, 8] demands complex static analysis. The mechanics involve relocating `m`, updating call sites, and adjusting accesses. Due to these complexities, existing solutions are incomplete; IDEs handle preconditions and mechanics, while research tools aim to identify opportunities.

The research community has proposed various approaches [2, 9–16] for identifying misplaced methods or recommending new target classes, typically optimizing software quality metrics like cohesion and coupling. These approaches fall into three categories: (i) static analysis-based [2, 9, 10], (ii) machine learning classifiers [11–13], and (iii) deep learning-based [14–16]. However, static analysis relies on expert-defined thresholds, and ML/DL methods require continual retraining as coding standards evolve, often diverging from real-world development practices.

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 refactorings that optimize software quality metrics are not always accepted in practice [17–26].

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. In our formative study, we found LLMs (GPT-4o, in particular) are prolific in generating suggestions, averaging 6 recommendations per class. However, two major challenges must be addressed to make this approach practical.

First, LLMs 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 (e.g., recommendations where the target class does not exist), and found that up to 80% of LLM recommendations are hallucinations. We discovered novel ways to automatically eliminate LLM hallucinations, by complementing 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*). We utilized code-trained vector embeddings from AI models to identify misplaced methods, and used refactoring preconditions [8] in existing IDEs (IntelliJ IDEA) to effectively remove *all* LLM hallucinations. We present these techniques in Section III-B.

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 context window of current LLMs [27]. Even with larger window sizes, passing the whole project as context introduces noise and redundancy, as not all classes are relevant; instead this further distracts the LLM [27, 28]. We address the limited context window of LLMs by using *retrieval augmented generation* (RAG) to enhance the LLM's input. Our two-step retrieval process combines mechanical feasibility (IDE-based static analysis) with semantic relevance (VoyageAI [29]), 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 III-C).

We designed, implemented, and evaluated these novel solutions as an IntelliJ IDEA plugin for Java, MM-ASSIST (Move Method 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 finally executes the correct refactoring based on user approval using the IDE.

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. We compare MM-ASSIST with the previous best in class approaches in their respective categories: JMOVE [10] – uses static analysis, FETRUTH [16] – uses Deep Learning, and HMOVE [30] – uses graph neural network to suggest moves and LLM to verify refactoring preconditions. These have been shown to outperform all previous MOVEMETHOD recommendation tools. Using a synthetic corpus widely used by previous approaches, we found that our tool significantly outperforms them: for class instance methods, our Recall@1 and Recall@3 are 67% and 75%, respectively, which is almost 1.75x 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 static methods. We compared against JMOVE, HMOVE and FETRUTH on this real-world oracle, and found that MM-ASSIST significantly outperforms them. Our Recall@3 is 80%, compared to 33% for the best baseline, HMOVE, a 2.4x improvement. This shows that MM-ASSIST's recommendations better align with developer best practices.

Whereas existing tools often require several hours to analyze a project and overwhelm developers with up to 57 recommendations to analyze per class, MM-ASSIST needs only about 30 seconds—even for tens of thousands of classes—and provides no more than 3 high-quality recommendations per class.

In a study where 30 experienced participants used MM-ASSIST on their own code for a week, 82.8% rated its 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 end-to-end LLM-powered assistant for MOVEMETHOD. Our approach advances key aspects: (i) recommendations are feasible and executed correctly, (ii) it requires no user-specified thresholds or model (re)-training, making it future-proof as LLMs evolve, and (iii) it handles both instance and static methods (avoided by others due to 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 window 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 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

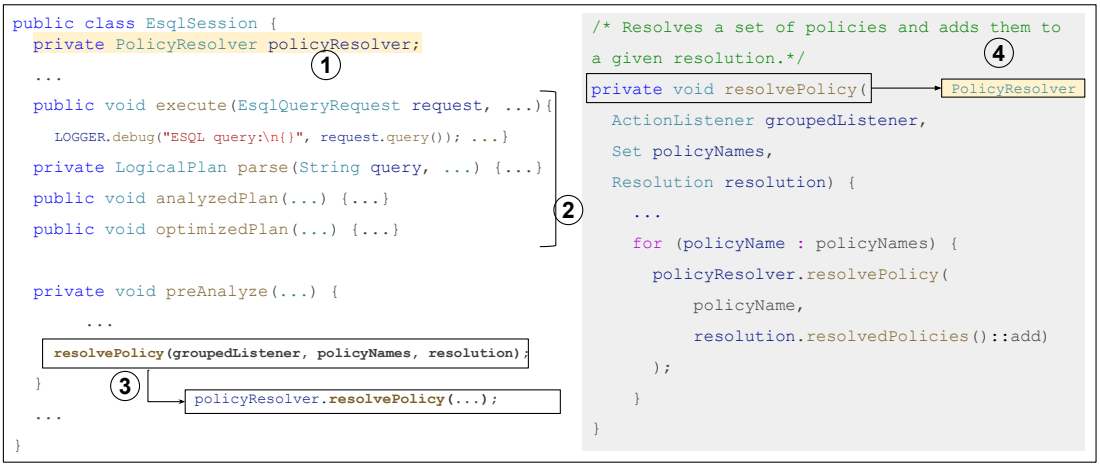

Fig. 1: A real-world example demonstrating a MOVEMETHOD on `resolvePolicy` performed by developers in the *Elasticsearch* project, commit 876e7015

improvements. Our user study confirms that MM-ASSIST aligns with and replicates real-world expert logic.

- **Replication.** We freely release MM-ASSIST, the datasets we used in the experiments, LLM prompts, demo, etc, so others can build upon these [31].

II. 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 [32]. 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.

Automatically identifying such refactoring opportunities is essential for maintaining software quality, but it poses significant challenges for existing tools. We first applied HMOVE [30], the state-of-the-art MOVEMETHOD technique. HMOVE, a classification tool, takes in <method, targetClass> pairs, and gives a probability score indicating whether to move the method. After computing all 158 pairs of inputs, HMOVE executed for 1.5 hours and generated 36 MOVEMETHOD recommendations. Its first and second highest recommendations to move are `subfields` and `execute`. Its third recommendation to move is `resolvePolicy`. However, HMOVE recommended moving `resolvePolicy` to the class `FunctionRegistry`, which is not an appropriate fit, as the method does not interact with it. This illustrates major shortcomings of classification-based tools like HMOVE: they need to be triggered on lots (avg. 145) of <method, targetClass> pairs, which means long runtime. Furthermore, they overwhelm the users with so many recommendations to analyze, testing the developer's patience and endurance. Moreover, they don't provide actionable steps.

Next, we ran JMOVE [10], a state-of-the-art MOVEMETHOD recommendation tool that solely relies on static analysis. To analyze the whole project JMOVE requires 12+ hours. To speed up JMOVE, we ran it on a sub-project of Elasticsearch containing `EsqlSession`. After it finished running for 30 minutes on a sub-project of Elasticsearch, JMOVE did not produce any recommendations for the `EsqlSession` class. This highlights a major shortcoming of existing static-analysis based tools like JMOVE: they need to analyze the entire project-structure and compute program dependencies – thus they do not scale to medium to large-size projects like `Elasticsearch` (800K LOC).

Next, we explored the potential of Large Language Models (LLMs) to recommend MOVEMETHOD refactoring. We used GPT-4o, a state-of-the-art LLM developed by OpenAI [33], and prompted it with the content of the `EsqlSession` class, asking: *"Identify methods that should move out of the `EsqlSession` class and where?"*. Our result 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 developers. Notably, the LLM did successfully identify `resolvePolicy` as a candidate for refactoring, showing its ability to detect semantically *misplaced* methods. Despite success, the LLM recommended other methods before `resolvePolicy`. A developer would need to filter out many irrelevant suggestions before arriving at a useful 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-fit semantically 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 [34] (even though context limits continuously increase). Even state-of-the-art LLMs can't process large projects like *Elasticsearch* in a single prompt without truncating crucial information. Moreover, the processing cost of 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, and the need for a tedious manual analysis. Developers need to manually collect and re-analyze the suggestions, verify the suitability of each method for relocation, prompt the LLM again for suitable target classes, 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 needs to sift through 5 candidate methods and, for each method, understand if any of the 5 or more proposed target classes are adequate. The developer analyzes 20+ <method, target class> pairs before finding one they agree with.

This example motivates our approach, MM-ASSIST, which significantly streamlines the refactoring process by (1) utilizing semantic relevance to find candidate 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 expertly recommends as the top candidate moving `resolvePolicy` to `PolicyResolver`. MM-ASSIST liberates developers so they can focus on the creative part. Rather than sifting through many invalid recommendations, developers use their expertise to examine a few high-quality recommendations.

III. APPROACH

Figure 2 shows the architecture and the steps performed by MM-ASSIST. First, MM-ASSIST applies a set of preconditions that filter out the methods that cannot be safely moved, such as constructors (①in Figure 2). It then leverages vector embeddings from Language Models to identify methods that are the least cohesive with their host class (②in Figure 2). In Figure 1, by comparing the embeddings of `resolvePolicy` and `EsqlSession` using cosine similarity, MM-ASSIST detects that this method might be misplaced (§ III-B). Next, MM-ASSIST passes the remaining candidates to the LLM (i.e., the method signature and the class body), which analyzes their relationships with the host class to prioritize the most promising MOVEMETHOD recommendations (③in Figure 2).

Once MM-ASSIST identifies candidate methods, it systematically evaluates potential target classes from the project codebase. For the `resolvePolicy` method, which utilizes the `enrichPolicyResolver` field (①in Figure 1), MM-ASSIST initially identifies several candidate classes, including `EnrichPolicyResolver` and `PolicyManager`.

To narrow down the target classes, MM-ASSIST calculates relevance scores between the candidate method and each potential target class – establishing a ranking (④in Figure 2). We label this process as "refactoring-aware RAG". Then, we feed the LLM with the narrowed-down list of target classes, and ask it to pick the best one (⑤in Figure 2). In this case, it correctly selects `EnrichPolicyResolver` as the appropriate destination for `resolvePolicy`, aligning with the developers' actual refactoring decision (see ③in Figure 1). Finally, refactoring suggestions (⑥in Figure 2) are presented to the user, and MM-ASSIST leverages the IDE's refactoring APIs to safely execute the chosen one automatically.

Next, we discuss each of these steps and concepts in detail.

A. Important Concepts

Definition III.1. (MOVEMETHOD Refactoring) *A* MOVEMETHOD *refactoring moves a method* m *from a host class* H *(where it currently resides) to a target class* T*. We define a* MOVEMETHOD *"*ω*" as a triplet* (m, H, T).

Definition III.2. (MOVEMETHOD Recommendations) *A list of* MOVEMETHOD *refactoring candidates, ordered by priority (most important at the beginning). We denote this with* $\Re$.

Definition III.3. (Valid Refactoring Recommendations) These are recommendations that do not break the code. They are mechanically feasible: they successfully pass the preconditions as checked by the IDE, thus resulting in syntactically and semantically equivalent code. We differentiate between moving an instance and a static method:

1) An **Instance Method** can be moved to a type in the host class' fields, or a type among the method's parameters. Several preconditions ensure the validity of the MOVEMETHOD recommendation, including:
   - Method Movability: Is the method a part of the class hierarchy? If not, the method can be safely moved.
   - Access to references: Does the moved method loses access to the references it needs for 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.

Definition III.4. (Invalid MOVEMETHOD Recommendations) We classify the LLM's invalid MOVEMETHOD suggestions as hallucinations and categorize them as follows:

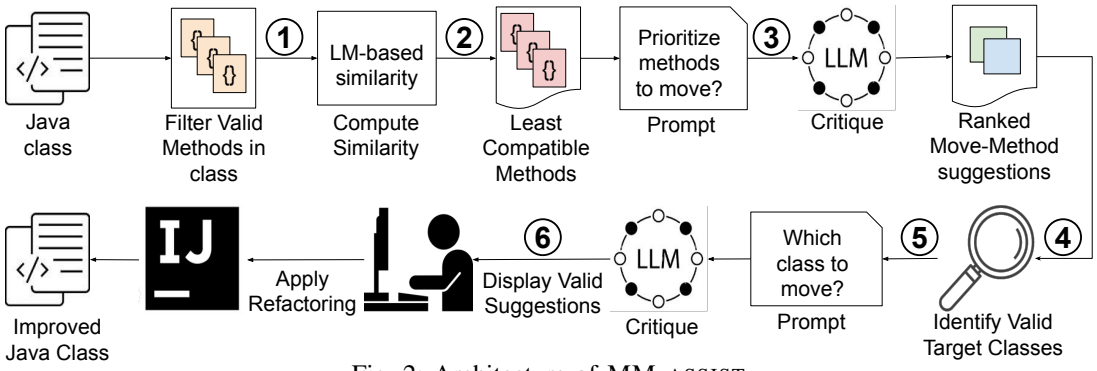

Fig. 2: Architecture of MM-ASSIST.

1) **Target class does not exist (H1):** The LLM comes up with an imaginary class.
2) **Mechanically infeasible to move (H2):** The target class exists, but a refactoring suggestion is invalid according to definitions in the previous subsection III.3.
3) **Invalid Methods (H3)**: The method is a part of the software design, and moving it requires multiple other refactoring changes. For example, moving a getter/setter needs to be accompanied by moving the appropriate field.

B. Identifying Which Method To Move

While we believe LLMs have great potential for suggesting MOVEMETHOD refactorings, directly using LLMs to identify potential methods that may be misplaced within a class is risky, as it results in invalid MOVEMETHOD recommendations.

Filter Invalid Candidates via Sanity Checks. Following established refactoring practices [10, 16, 30], we filter 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 subclasses, since moving these would require additional structural changes. It also removes test methods and those with irrelevant content, such as empty bodies or methods containing only comments.

Identifying Semantically Mismatched Methods. For the remaining candidates, the next step is to find methods that are out of place in their current class. To automate this, we use code embeddings (VoyageAI [29]) to capture the semantic meaning of the code. An embedding is a vector representation that captures the semantic features of an entity (methods and classes) based on its content and relationships. We use VoyageAI due to its state-of-the-art performance in code-related tasks. We generate vectors 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.

Prioritizing Candidates with LLM-driven Reasoning. Finally, after identifying a pool of semantically misplaced methods, we use an LLM to rank them based on a broader perspective (i.e., class level design). We use the LLM to rank the existing methods in our suggestion pool. Using Chain-of-Thought (CoT) reasoning, we prompt the LLM to perform a structured analysis: evaluate each method's purpose, cohesion and dependencies, summarize the host class's responsibilities, and assess overall alignment of the method & class.

C. 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) [35]. RAG is a systematic approach designed to retrieve and refine relevant contextual information, thereby augmenting the input to the LLM. In MM-ASSIST, we efficiently retrieve and augment the model with the most relevant target classes. Instance methods can be moved to a limited number of feasible classes. Static methods can be moved to almost any class in the project. Thus, we compare the structure and semantics to find the most suitable target classes (described below). In both cases, we rank target classes based on semantic relevance and finally provide a suitable list of target classes to the LLM to allow it to choose the best fit. Since we designed the retrieval process to enhance refactoring, we call this "refactoring-aware RAG", and we explain it below.

1) Candidate Filtering by Method Type: The initial filtering process differs based on whether the method is an instance or static type, as they possess different relocation constraints.

*a) **Target Class Retrieval for Instance Methods**:* As instance methods can be moved to a few suitable classes, we select the method's parameter types and the host class' field types as potential destinations. Then, we utilize the IDE's preconditions to retain only valid MOVEMETHOD suggestions.

*b) **Target Class Retrieval for Static Methods**:* We identify potential target classes within the project based on two key aspects: package proximity and utility class identification. Package proximity quantifies structural closeness in the package hierarchy by computing shared package segments (e.g., for org.example.core and org.example.utils, "org" and "example" are shared) normalized by the host package depth, providing

an initial structural filter. Utility classes, identified through conventional naming patterns (containing "util" or "utility"), are prioritized as common targets for static methods. These heuristics are efficient filters to narrow down the search space of potential target classes. We rank potential target classes based on the above heuristics, with greater importance given to proximity. Our ranking function is:

$$RankingScore(\texttt{T}) = 2 \cdot \text{proximity}(\texttt{T}, \texttt{H}) + \text{isUtility}(\texttt{T}) \quad (1)$$

where:

- proximity($\texttt{T}, \texttt{H}$) evaluates the package proximity between class `T` and the host class `H`
- isUtility($\texttt{T}$) is a boolean function that returns 1 if `T` is a utility class, and 0 otherwise

Finally, we utilize static analysis from the IDE to validate whether the method can be moved to the potential target class.

2) Semantic Relevance-Based Target Class Ranking: While static analysis offers foundational understanding of valid refactoring opportunities, it often yields a broad set of potential target classes, as it lacks the ability to capture deeper semantic relationships. To augment the results of static analysis, we incorporate a semantic relevance analysis, which compares both the content and intent of the candidate method and target classes. To do this, we use VoyageAI's vector embeddings [29] to compute the cosine similarity between the method body and potential target classes. We sort the target classes by their cosine similarity scores in descending order to select the most semantically relevant candidates for the LLM to analyze. To stay within the LLM's context window, we limit the candidates to those fitting within a 7K token budget – typically accommodating 10-12 class summaries with their signatures.

3) Ranking Target Classes Using LLM: In the final phase, MM-ASSIST asks the LLM for the best-suited target class, utilizing its vast training knowledge. To avoid context overflow, we create a concise representation of each target class, including its name, field declarations, DocString, 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.

D. Applying Refactoring Changes

After compiling a list of MOVEMETHOD recommendations, MM-ASSIST presents the 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. It then executes the command automatically through the IDE's refactoring APIs, ensuring safe code transformation (moving the method, and changing all call sites and references).

IV. EMPIRICAL EVALUATION

To evaluate MM-ASSIST's effectiveness and usefulness, we designed a comprehensive, multi-methodology evaluation to corroborate, complement, and expand our findings. This includes a formative study, comparative study, replication of real-world refactorings, repository mining, user study, and questionnaire surveys. These methods combine qualitative and quantitative data, and together answer four research questions.

RQ1. How effective are LLMs at suggesting opportunities for MOVEMETHOD refactoring? This question assesses Vanilla LLMs' ability to recommend MOVEMETHOD in a formative study examining the quality of LLM suggestions.

RQ2. How effective is MM-ASSIST at suggesting opportunities for MOVEMETHOD refactoring? We evaluate the performance of MM-ASSIST against the state-of-the-art tools, FETRUTH [16] and HMOVE [30] (representatives for DL approaches), and JMOVE [10] (representative for static analysis approaches). We used both a synthetic corpus used by other researchers and a new dataset of real refactorings from 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.

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 our tool on their own code for a week.

A. Subject Systems

To evaluate LLMs' capability when suggesting MOVEMETHOD refactoring opportunities, we employed two distinct datasets: a synthetic corpus widely used by previous researchers [10, 13, 30, 36] 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 III.1) - each containing a triplet of method-to-move, host class, and target class $(\texttt{m}, \texttt{H}, \texttt{T})$.

1) Synthetic corpus: The *synthetic corpus* was created by Terra et al. [10] moving different methods `m` out of their original/host class `H` to a random destination class `T`. The researchers then created the gold set as tuples $(\texttt{m}, \texttt{H}, \texttt{T})$, i.e., methods `m` that a tool should now move from `H` back to its original class `T`. This dataset moves *only instance methods*; it does not move *static methods*. This corpus consists of 10 open-source projects.

2) Real-world corpus: As refactorings in the real-world are often complex and messy [37], we decided to complement the synthetic dataset with a corpus of *actual* MOVEMETHOD refactorings that open-source developers performed on their projects. This dataset allows us to determine whether various tools can match the rationale of expert developers in real-world situations. We construct this oracle using RefactoringMiner [38], the state-of-the-art for detecting refactorings [39].

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.

However, many of the MOVEMETHOD reported by RefactoringMiner are not pure MOVEMETHOD refactorings, 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) [40]. These impure MOVEMETHOD operations require additional changes to the code's structure, to facilitate them (e.g. creating a new class, moving fields, etc.). To deal with the limited number of data points from MOVEMETHOD refactorings detected by RefactoringMiner, we leveraged 'Extract and Move Method' refactoring detected by RefactoringMiner (a composite refactoring in which developers first extract a new method and then move it to a different class). Empirically, we found that detected 'Extract and Move Method's resulted in a greater number of pure refactorings than detected MOVEMETHOD. This can be attributed to the reason that small sections of a methods are out of place, rather than the entire method, and performing the extract highlights the need for move. Moreover, the 'Extract and Move Method' refactoring is frequently performed in practice. We constructed our dataset by executing the extract method (within the IDE), and generating an intermediate version of the code for tools to analyze and generate recommendations. Then, we performed further analysis to validate that the target method can in-fact be moved to the target-class. To achieve this, we applied a process similar to prior work [16, 30]: 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). We removed test methods, getters, setters, and overriden methods (they violate preconditions for a MOVEMETHOD). 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 signature of the moved method contained a reference to the source class. 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 popular open-source projects. On average, each project contains 8743 classes and 66306 methods spanning 1032344 LOCs. This oracle enables an evaluation of MM-ASSIST on authentic refactorings made by experienced developers.

B. Effectiveness of LLMs (RQ1)

1) Evaluation Metrics: Using these datasets, we evaluated the recommendations made by the vanilla LLM and identified the hallucinations, as defined in Definition III.4: H1 – target class does not exist in the project; H2 – moving the method to the target class is mechanically infeasible; and H3 – violating preconditions in Section III-B.

2) Experimental Setup: We use the vanilla implementation of GPT-4o, a state-of-the-art LLM from OpenAI [33]. We used a version that was released on May 13, 2024. While MM-ASSIST is model agnostic (i.e., we can simply swap different models), we chose GPT-4o because researchers [41, 42] show that it outperforms other LLMs when used for refactoring tasks. GPT-4o is also widely adopted in many software-engineering tools [43–46]. We designed our experimental setup to assess the model's inherent capabilities in understanding and recommending MOVEMETHOD without additional context or task-specific tuning. We formulated a prompt where we provided the source code in a given host class and asked the LLM which methods to move and where. We set the LLM temperature parameter to 0 to obtain deterministic results.

For each host class in our Gold sets (synthetic and real-world), we submitted prompts to the LLM and collected its recommendations.

TABLE I: Different kinds of hallucinations from Vanilla LLM

Corpus	# R	# H1	# H2	# H3
Synthetic (235)	723	362	168	51
Real-world (210)	1293	431	275	320

R: Recommendations, H1: Hall-class, H2: Hall-Mech, H3: Invalid Method.

3) Results: Table I 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 a prevalence in all three types of hallucinations: H1, H2, and H3 (as defined in Definition III.4) *Crucially, actuating any of these hallucinations would lead to broken code, compilation errors, or degraded software design.*

In both the synthetic and real-world dataset, a mere 20% (142/723 and 267/1293 respectively) were valid. The overwhelming 80% were hallucinations. These findings underscore the impracticality of using vanilla LLM recommendations for MOVEMETHOD without extensive filtering and validation. For every valid recommendation, a developer would need to sift through and discard 3-4 invalid ones – which may introduce critical errors if implemented. This undermines the potential time-saving benefits of automated refactoring and introduces significant risks of introducing bugs or degrading code quality.

C. Effectiveness of MM-ASSIST (RQ2)

To evaluate MM-ASSIST's effectiveness, we compared it against state-of-the-art MOVEMETHOD recommendation tools.

1) Baseline Tools: We directly compare with the best-in-class tools: JMOVE [10] is a state-of-the-art static analysis tool, FETRUTH [16] is the former best in class tool that uses ML/DL, and HMOVE [30] is a recently introduced state-of-the-art tool that uses graph neural networks to generate suggestions and LLM to check refactoring preconditions. HMOVE has 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 the most fair comparison: we consulted with HMOVE, 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. We are grateful for their assistance. Further, we also replicated the recall numbers presented in each tool's paper, validating our experimental setup.

2) Evaluation Metrics: For evaluation, we employ recall-based metrics, following an approach similar to that used in the evaluation of JMove [10], a well-established tool in this domain. In our setting, recall is a more suitable metric (against precision) because it measures how many relevant recommendations are retrieved, avoiding the need for subjective judgments about whether a recommendations is a false positive. Furthermore, to provide actionable recommendations and avoid overwhelming the developer, we use recall@k, which returns a small number of recommendations (k). Furthermore, recall@k is similar to precision by evaluating the quality of a limited set of recommendations.

We present recall for each phase of suggesting the move-method refactoring: first, identifying that a method is misplaced, no matter the recommened target class ($Recall_M$); 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 $\Re$ (see III.2), we define $Recall_M$, $Recall_C$ and $Recall_{MC}$ as follows:

$$\mathbf{Recall_M} = \frac{|\Re_M|}{|G|},\ \mathbf{Recall_C} = \frac{|\Re \cap G|}{|\Re_M|},\ \mathbf{Recall_{MC}} = \frac{|\Re \cap G|}{|G|}$$

Where $\Re_M$ is the subset of $\Re$ containing refactorings whose method components match those in the ground truth set G. Formally, we define $\Re_M$ as follows:

$$\Re_M = \{g | g \in G \wedge \exists (g_m, g_c, *) \in \Re\}$$

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

3) Experimental Setup: We trigger all tools on each host class from the gold set (both synthetic and the real-world dataset). To account for the inherent non-determinism in LLMs, we ran the vanilla LLM with multiple temperature values (0, 0.5, 1) for five runs each. Additionally, we ran MM-ASSIST three times. We report the average and standard deviation for both vanilla LLM and MM-ASSIST. 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. HMOVE takes both the method and candidate target class as input, and returns a probability score indicating whether to move the method. As a result, it becomes impractical to recommend moving static methods, because there can be thousands of (method, target-class) pairs – each taking a minute on average to classify. For example, to recommend moving a single static-method in the Elasticsearch project with 21615 target classes would require $\approx$ 14 days to process. Thus, we limit HMOVE to instance move method recommendations. We do not penalize any tool for running out of time – we do not compute recall rates on those data points. For example in Table 3, JMOVE only finished computing for 24 out of 70 Large Classes, so we only report JMOVE 's Recall for those 24 Classes (i.e., we do not compute nor report a zero recall for the 46 Classes it ran out of time).

All the tools generate a ranked list of MOVEMETHOD suggestions. We compared these suggestions against the ground truth to calculate recall for each tool using the evaluation metrics presented earlier – $Recall_M$, $Recall_C$, and $Recall_{MC}$.

4) Results on the Synthetic Corpus: Table II shows the effectiveness of MM-ASSIST and baseline tools on the synthetic dataset. MM-ASSIST demonstrates superior performance across many of the recall metrics compared to other tools, especially $Recall_{MC}$@1 (1.7x to 30x improvement) – the most comprehensive measure assessing both correct method and target class identification. While JMOVE exhibits high accuracy in target class identification ($Recall_C$@1 = 97%), it shows limitations in method identification. HMOVE demonstrates comparable performance as JMOVE in identifying misplaced methods (i.e., $Recall_M$) but does not match JMOVE's ability to recommend the correct target class (i.e., $Recall_C$). Consequently, its combined $Recall_{MC}$ is lower than JMOVE. 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. When it does correctly identify a method, it accurately suggests the target class. However, because it rarely identifies the misplaced methods themselves, the overall $Recall_{MC}$ remains low. We confirmed this paradox with FETRUTH authors. Interestingly, the Vanilla-LLM shows comparable performance to MM-ASSIST in method identification ($Recall_M$). This could be because the LLMs have been pre-trained on the synthetic dataset, and might have memorized their responses – thus becoming necessary to use an LLM uncontaminated dataset (see next).

5) Results on the Real-world Corpus: With the real-world dataset performed by open-source developers, we found a wider difference in performance. First, we distinguish between cases when the MOVEMETHOD target was an instance or static method – shedding light on the effectiveness of MM-ASSIST in different usage scenarios. Second, we distinguish between small and large classes based on their method count. Our analysis of the real-world oracle reveals a heavy-tail distribution – we label classes with fewer than 15 methods (90th percentile across all projects) as Small Classes (avg. 6 methods/class) and the rest as Large Classes (avg. 48 methods/class).

Tables III and IV summarize our results for instance and static method moves, respectively. Since JMOVE could not finish running sometimes and HMOVE sometimes failed with a syntax error due to unsupported Java language features, we note the number of completed entries in parenthesis. For instance methods in Small Classes, MM-ASSIST achieved 2.4x to 4x ($Recall_{MC}$@3) higher recall compared to baseline tools. For Small Classes, we noticed that our performance 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, we observed a performance degradation in all tools when identifying MOVEMETHOD opportunities in large classes. This happens because large classes are more prone to significant technical debt, and there are many candidate methods that can be moved – thus it is harder to pick the proverbial "needle from the haystack".

However, the differences are more nuanced when we evalu-

TABLE II: Recall rates of MM-ASSIST on the synthetic corpus of 235 refactorings [10] that moved instance methods. $Recall_M$ = identify the method, $Recall_C$ = identify the target class for a 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%	43%	43%	97%	97%	97%	40%	42%	42%
FETRUTH	2%	3%	3%	**100%**	**100%**	**100%**	2%	3%	3%
HMOVE	31%	37%	40%	32%	37%	39%	21%	24%	26%
Vanilla-LLM	71±2%	75±1%	79±2%	70±1%	70±1%	70±1%	53±2%	55±2%	57±2%
MM-ASSIST	**72±1%**	**79±0%**	**80±0%**	91±1%	97±0%	98±1%	**67±0%**	**73±0%**	**75±0%**

TABLE III: 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%
	HMOVE (30)	23%	37%	40%	37%	43%	47%	17%	30%	33%
	Vanilla-LLM	52% ±3	71% ±3	83% ±3	67% ±4	67% ±4	67% ±4	58% ±3	54% ±2	58% ±3
	MM-ASSIST	**75% ±1**	**92% ±0**	**95% ±0**	85% ±1	89% ±0	89% ±0	**68% ±2**	**80% ±1**	**80% ±1**
LargeClasses (70)	JMOVE (24)	8%	8%	8%	**100%**	**100%**	**100%**	8%	8%	8%
	FETRUTH	2%	8%	12%	76%	76%	76%	2%	6%	9%
	HMOVE (55)	9%	15%	20%	15%	20%	24%	4%	5%	5%
	Vanilla-LLM	15% ±2	23% ±2	24% ±2	44% ±8	44% ±8	44% ±8	8% ±1	8% ±1	8% ±1
	MM-ASSIST	**36% ±1**	**38% ±1**	**47% ±1**	75% ±1	80% ±1	80 ±2	**30% ±1**	**31% ±2**	**36% ±2**

TABLE IV: 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	42% ±1	55% ±2	64% ±1	8% ±1	8% ±1	8% ±1	4% ±1	4% ±1	4% ±1
	MM-ASSIST	**54% ±1**	**62% ±3**	**69% ±1**	**22% ±4**	**26% ±4**	**26% ±4**	**12% ±2**	**16% ±2**	**18% ±2**
LargeClasses (62)	FETRUTH	6%	11%	15%	6%	6%	6%	0.4%	1%	1%
	Vanilla-LLM	14% ±1	22% ±1	28% ±6	3% ±5	3% ±5	3% ±5	0.3% ±1	1% ±1	1% ±2
	MM-ASSIST	**17% ±4**	**28% ±4**	**32% ±4**	**43% ±5**	**43% ±5**	**47% ±5**	**7% ±1**	**12% ±1**	**15% ±1**

ate 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 a tool should analyze thousands of classes to find the right one. 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.

D. Runtime performance of MM-ASSIST *(RQ3)*

1) Experimental Setup: We used both the synthetic and real-world corpus employed in other RQs to measure the time taken for each tool to produce recommendations. 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. We conducted all experiments on a commodity laptop, an M1 MacBook Air with 16GB of RAM.

2) 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. Similarly, HMOVE also takes an average of 80min to execute on a single entry in our dataset – it needs to be triggered on all possible <method, target class> pairs (avg. 145 pairs per class). In extreme cases where the host class was large (>10K LOC), HMOVE took 4 whole days to execute on a single entry in our dataset. Out of the box, FETRUTH is also slow and can take 12+ hours to run on large projects. With the help of FETRUTH authors, we were able to run it on a single class at a time – this takes an average 6 minutes per class. Thus, compared with the baselines, MM-ASSIST is two, two, and one order(s) of magnitude faster than JMOVE, HMOVE, and FETRUTH, respectively. Thus, it is practical.

E. Usefulness of MM-ASSIST (RQ4)

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

1) 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, familiarity with their codebases 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.

2) 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 their projects. 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 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.

The user study involved setting up our IntelliJ plugin, executing our tool on a class to get refactoring suggestions, rating the suggestions live inside the IDE, and lastly, providing feedback by filling out our survey. 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. 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 scale 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. 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 asked participants to compare MM-ASSIST's workflow against the IDE, and asked for open-ended feedback about their experience.

3) Results: 30 participants applied MM-ASSIST on 350 classes. MM-ASSIST analyzed 1,143 host classes where developers did not have prior knowledge of refactoring opportunities. MM-ASSIST recommended refactorings in 350 of them, with an average of 1.7 recommendations/class. MM-ASSIST did not deem 793 typical classes to require refactoring, avoiding unnecessary developer effort. We found that, in 290 classes the participants positively rated one of the recommendations (82.8% of the time). Moreover, the users accepted and applied a total of 216 refactoring recommendations on their code (out of 354 total recommendations), i.e. 7 refactorings per user, on average. This shows that our tool 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.

V. 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 generalization of our results. Because we rely on a specific LLM (GPT-4o), 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- and refactoring-agnostic, extending to additional refactoring types and languages requires adapting three key components: (1) static analysis for validating refactoring preconditions, (2) semantic analysis of code relationships, and (3) refactoring execution mechanics. Using protocols like the Language Server Protocol (LSP) [47] can simplify handling language-specific features, facilitating broader applicability. Future work will explore the effectiveness of our tool across various languages and refactorings. While we utilize IntelliJ's refactoring engine for validation, we acknowledge it may contain latent bugs [48]. To mitigate this, MM-ASSIST employs a sanity check to pre-filter invalid move methods. Therefore, the reliability of our approach depends not only on the correctness of IntelliJ's engine but also on the effectiveness of our pre-filtering stage.

Tool implementation. MM-ASSIST's implementation follows a modular architecture, separating language-specific concerns from the core refactoring workflow. Components such as the LLM service, embedding model, and IDE integration communicate via well-defined interfaces, facilitating extensibility and integration across various environments and languages. To address the non-deterministic nature of LLMs, we experimented with various temperature settings and found the variability in the LLM's outputs to be consistently low, as evidenced by the small standard deviations reported in our

evaluation. For IDE developers, MM-ASSIST shows the safe integration of AI-powered suggestions with existing IDEs.

VI. Related Work

We organize the related work into: (i) research on MOVEMETHOD, and (ii) usage of LLMs for refactoring.

MOVEMETHOD refactoring. Many researchers focus on identifying and recommending MOVEMETHOD refactorings. JMOVE [10], JDeodorant [49], and MethodBook [9] suggest refactorings based on software metrics derived from static analysis. Additionally, JMOVE introduced a widely-used synthetically created dataset of MOVEMETHOD refactorings. HMOVE [30], a recently introduced tool, uses graph neural networks to classify a MOVEMETHOD suggestion as go/no-go. Then, HMOVE only uses LLM as a judge to filter suggestions that don't meet certain preconditions. Similarly, FETRUTH [16], RMove [13], and PathMove [36], utilize DL techniques to identify MOVEMETHOD opportunities.

Most importantly, MM-ASSIST attacks the problem in a different direction. 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*, not a *recommendation* problem: they produce 57 recommendations on average to move out a given class (many of which are unuseful), which puts tremendous analysis burden on programmers. In contrast, MM-ASSIST offers at most 3 recommendations per class, aligned with how expert developers refactor code.

Refactoring in the age of LLMs. A recent systematic study [50] 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 [30, 51–54] and detect code smells [55]. Cui et al. [56] leverage intra-class dependency hypergraphs with LLMs to perform extract class refactoring, while iSMELL [57] uses LLMs to detect code smells and suggest corresponding refactorings. However, LLMs are prone to hallucinate, which can introduce incorrect or broken code, posing challenges for automated refactoring systems. Unlike other approaches, MM-ASSIST 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. [41, 58] investigated hallucinations in EXTRACTMETHOD refactoring, while [42] 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 MM-ASSIST, ensuring the reliability and safety of the suggestions generated by LLMs. MANTRA [59] introduces a multi-agent approach with RAG to automate refactoring. However, MANTRA uses LLM to generate the refactored code directly, guiding this process with a RAG system that retrieves examples of correct refactorings from open-source projects to serve as few-shot prompts. In contrast, we leverage the LLM only to identify and propose refactoring candidates, while delegating the safe execution of the code transformation to the IDE. Our application of RAG also differs, as we use it to narrow down the search and retrieve a smaller number of potential target classes.

VII. Conclusion

Despite years of research in MOVEMETHOD refactoring, progress has been incremental. The rise of LLMs has revitalized 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 refactorings from recent open-source projects, MM-ASSIST achieves 4x higher recall while running 10x–100x faster. Additionally, in a one-week case study, 30 experienced developers rated 82.8% of MM-ASSIST 's 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. By leveraging semantic embedding into a RAG approach, MM-ASSIST narrows down the context for the LLM so that it can focus on a small number of high-quality prospects. This was instrumental in picking the right candidate from industrial large scale projects. We hope that these techniques inspire others to solve other refactoring recommendation domains, e.g., splitting large classes or packages.

VIII. Data Availability

To ensure the verifiability of our work and to provide a foundation on which others in the community can build upon, we have made our tool, MM-ASSIST, publicly available in our replication package [31]. This includes the complete source code, the IntelliJ plugin implementation, comprehensive documentation detailing the setup and usage instructions, a demo of MM-ASSIST in action, the exact LLM prompts that we use, and all datasets we use in our evaluation.

Acknowledgements

We are grateful for the constructive feedback from the members of the AI Agents team at JetBrains Research and the anonymous conference reviewers. This research was partially funded through the US National Science Foundation (NSF) grants CNS-1941898, CNS-2213763, 2512857, 2512858, the Industry-University Cooperative Research Center on Pervasive Personalized Intelligence (PPI), and a gift grant from NEC. Tien N. Nguyen was supported in part by the NSF grant CNS-2120386, and the National Security Agency (NSA) grant NCAE-C-002-2021.

References

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

[2] N. Tsantalis and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities," *TSE*, 2009.

[3] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures," *JSS*, 2011.

[4] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *FSE*, 2016.

[5] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A Comparative Study of Manual and Automated Refactorings," in *ECOOP*, 2013.

[6] E. Murphy-Hill, C. Parnin, and A. P. Black, "How We Refactor, and How We Know It," *TSE*, 2012.

[7] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *TSE*, 2022.

[8] W. F. Opdyke, "Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[9] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending Move Method Refactorings via Relational Topic Models," *TSE*, 2014.

[10] R. Terra, M. T. Valente, S. Miranda, and V. Sales, "JMove: A novel heuristic and tool to detect move method refactoring opportunities," *JSS*, 2018.

[11] T. Bryksin, E. Novozhilov, and A. Shpilman, "Automatic recommendation of move method refactorings using clustering ensembles," in *IWoR*, 2018.

[12] Z. Kurbatova, I. Veselov, Y. Golubev, and T. Bryksin, "Recommendation of Move Method Refactoring Using Path-Based Representation of Code," in *ICSEW*, 2020.

[13] D. Cui, S. Wang, Y. Luo, X. Li, J. Dai, L. Wang, and Q. Li, "RMove: Recommending Move Method Refactoring Opportunities using Structural and Semantic Representations of Code," in *ICSME*, 2022.

[14] H. Liu, Z. Xu, and Y. Zou, "Deep learning based feature envy detection," in *ASE*, 2018.

[15] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu, and L. Zhang, "Deep Learning Based Code Smell Detection," *TSE*, 2021.

[16] B. Liu, H. Liu, G. Li, N. Niu, Z. Xu, Y. Wang, Y. Xia, Y. Zhang, and Y. Jiang, "Deep Learning Based Feature Envy Detection Boosted by Real-World Examples," in *ESEC/FSE*, 2023.

[17] J. Ivers, A. Ghammam, K. Gaaloul, I. Ozkaya, M. Kessentini, and W. Aljedaani, "Mind the Gap: The Disconnect Between Refactoring Criteria Used in Industry and Refactoring Recommendation Tools," in *ICSME*, 2024.

[18] A. C. Bibiano, W. K. G. Assunção, D. Coutinho, K. Santos, V. Soares, R. Gheyi, A. Garcia, B. Fonseca, M. Ribeiro, D. Oliveira, C. Barbosa, J. L. Marques, and A. Oliveira, "Look Ahead! Revealing Complete Composite Refactorings and their Smelliness Effects," in *ICSME*, 2021.

[19] L. P. Antonio Mastropaolo, Emad Aghajani and G. Bavota, "Automated variable renaming: are we there yet?" in *EMSE*, 2023.

[20] J. Pantiuchina, B. Lin, F. Zampetti, M. Di Penta, M. Lanza, and G. Bavota, "Why Do Developers Reject Refactorings in Open-Source Projects?" *TOSEM*, 2021.

[21] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? Confessions of GitHub contributors," in *FSE*, 2016.

[22] M. Dilhara, A. Ketkar, and D. Dig, "Understanding Software-2.0: A Study of Machine Learning Library Usage and Evolution," *TOSEM*, 2021.

[23] M. Dilhara, A. Ketkar, N. Sannidhi, and D. Dig, "Discovering repetitive code changes in Python ML systems," in *ICSE*, 2022.

[24] M. Dilhara, D. Dig, and A. Ketkar, "PYEVOLVE: Automating Frequent Code Changes in Python ML Systems," in *ICSE*, 2023.

[25] S. Fakhoury, D. Roy, A. Hassan, and V. Arnaoudova, "Improving source code readability: Theory and practice," in *ICPC*, 2019.

[26] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability," *TSE*, 2019.

[27] S. Tworkowski, K. Staniszewski, M. a. Pacek, Y. Wu, H. Michalewski, and P. Mił oś, "Focused Transformer: Contrastive Training for Context Scaling," in *NeurIPS*, 2023.

[28] "Needle In A Haystack - Pressure Testing LLMs," https://github.com/gkamradt/LLMTest_NeedleInAHaystack.

[29] VoyageAI, "Voyage AI Embeddings," https://docs.voyageai.com/docs/embeddings.

[30] D. Cui, J. Wang, Q. Wang, P. Ji, M. Qiao, Y. Zhao, J. Hu, L. Wang, and Q. Li, "Three Heads Are Better Than One: Suggesting Move Method Refactoring Opportunities with Inter-class Code Entity Dependency Enhanced Hybrid Hypergraph Neural Network," in *ASE*, 2024.

[31] "MM-ASSIST replication package," https://cuboulder-se-research.github.io/move-method-assist/.

[32] Elasticsearch, "Move Method Refactoring in the Elasticsearch Project," https://github.com/elastic/elasticsearch/commit/876e70159c01ae306251281ae2fdbabca8732ed9.

[33] "OpenAI," https://openai.com/.

[34] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the Middle: How Language Models Use Long Contexts," *TACL*, 2023.

[35] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *NeurIPS*, 2020.

[36] Z. Kurbatova, I. Veselov, Y. Golubev, and T. Bryksin, "Recommendation of Move Method Refactoring Using Path-Based Representation of Code," in *ICSEW*, 2020.

[37] D. Oliveira, W. K. G. Assunção, A. Garcia, A. C. Bibiano, M. Ribeiro, R. Gheyi, and B. Fonseca, "The untold story of code refactoring customizations in practice," in *ICSE, 2023*.

[38] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and Efficient Refactoring Detection in Commit History," in *ICSE*, 2018.

[39] O. Leandro, R. Gheyi, L. Teixeira, M. Ribeiro, and A. Garcia, "A Technique to Test Refactoring Detection Tools," in *SBES*, 2022.

[40] E. Novozhilov, I. Veselov, M. Pravilov, and T. Bryksin, "Evaluation of move method refactorings recommendation algorithms: Are we doing it right?" in *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, 2019, pp. 23–26.

[41] D. Pomian, A. Bellur, M. Dilhara, Z. Kurbatova, E. Bogomolov, T. Bryksin, and D. Dig, "Next-Generation Refactoring: Combining LLM Insights and IDE Capabilities for Extract Method," in *ICSME*, 2024.

[42] M. Dilhara, A. Bellur, T. Bryksin, and D. Dig, "Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example," in *FSE*, 2024.

[43] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, H. Zhang, B. Zhu, M. Jordan, J. E. Gonzalez, and I. Stoica, "Chatbot arena: An open platform for evaluating LLMs by human preference," 2024.

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

[45] Cursor, "Cursor," https://www.cursor.com/, accessed: 2024-10-07.

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

[47] "Language server protocol (lsp)," 2024, https://github.com/python-rope/pylsp-rope.

[48] H. Wang, Z. Xu, H. Zhang, N. Tsantalis, and S. H. Tan, "Towards understanding refactoring engine bugs," *ACM Transactions on Software Engineering and Methodology*, 2025.

[49] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Ten years of JDeodorant: Lessons learned from the hunt for smells," in *SANER*, 2018.

[50] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large Language Models for Software Engineering: A Systematic Literature Review," *TOSEM*, 2024.

[51] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe, "Refactoring programs using Large Language Models with few-shot examples," in *APSEC*, 2023.

[52] K. DePalma, I. Miminoshvili, C. Henselder, K. Moss, and E. A. AlOmar, "Exploring ChatGPT's code refactoring capabilities: An empirical study," *Expert Systems with Applications*, 2024.

[53] E. A. AlOmar, A. Venkatakrishnan, M. W. Mkaouer, C. Newman, and A. Ouni, "How to refactor this code? An exploratory study on developer-ChatGPT refactoring conversations," in *MSR*, 2024.

[54] B. Liu, Y. Jiang, Y. Zhang, N. Niu, G. Li, and H. Liu, "Exploring the potential of general purpose LLMs in automated software refactoring: an empirical study," *ASE(J)*, 2025.

[55] L. L. Silva, J. R. d. Silva, J. E. Montandon, M. Andrade, and M. T. Valente, "Detecting Code Smells using ChatGPT: Initial Insights," in *ESEM*, 2024.

[56] D. Cui, Q. Wang, Y. Zhao, J. Wang, M. Wei, J. Hu, L. Wang, and Q. Li, "One-to-One or One-to-Many? Suggesting Extract Class Refactoring Opportunities with Intra-class Dependency Hypergraph Neural Network," in *ISSTA*, 2024.

[57] D. Wu, F. Mu, L. Shi, Z. Guo, K. Liu, W. Zhuang, Y. Zhong, and L. Zhang, "iSMELL: Assembling LLMs with Expert Toolsets for Code Smell Detection and Refactoring," in *ASE*, 2024.

[58] D. Pomian, A. Bellur, M. Dilhara, Z. Kurbatova, E. Bogomolov, A. Sokolov, T. Bryksin, and D. Dig, "EM-Assist: Safe Automated ExtractMethod Refactoring with LLMs," in *FSE*, 2024.

[59] Y. Xu, F. Lin, J. Yang, N. Tsantalis *et al.*, "Mantra: Enhancing automated method-level refactoring with contextual RAG and multi-agent LLM collaboration," *arXiv preprint arXiv:2503.14340*, 2025.