Enhancing LLMs with Staged Grouping and Dehallucination for Header File Decomposition

Yue Wang
Key Lab of HCST (PKU), MOE;
SCS, Peking University
Beijing, China
wangyue0502@pku.edu.cn

Jiaxuan Sun
Key Lab of HCST (PKU), MOE;
SCS, Peking University
Beijing, China
mandala@stu.pku.edu.cn

Yanzhen Zou*
Key Lab of HCST (PKU), MOE;
SCS, Peking University
Beijing, China
zouyz@pku.edu.cn

Bing Xie*
Key Lab of HCST (PKU), MOE;
SCS, Peking University
Beijing, China
xiebing@pku.edu.cn

***Abstract*—God Header Files, large header files included by numerous other code files, present significant challenges for code comprehension and maintenance. Existing approaches leverage various code similarity metrics to decompose them, but these metrics do not always capture the code's functional essence accurately. Large Language Models (LLMs), with their advanced capabilities in code understanding and generation, offer a promising alternative for producing more effective refactorings. However, LLMs face three critical limitations that hinder practical application: they struggle with lengthy header files due to token constraints, suffer from hallucination by generating incomplete or spurious results, and produce cyclic dependencies that violate architectural principles and cause compilation failures. To address these challenges, we propose *HFDecomposer*, a hybrid approach that enhances LLMs with staged grouping and dehallucination techniques to effectively decompose header files. Our approach introduces a two-stage grouping framework for lengthy header files: it first groups strongly related code entities using traditional similarity metrics, then feeds group summaries to the LLM for higher-level semantic aggregation. To mitigate LLM hallucinations, we enhance prompts with factual knowledge extracted from static analysis, detect errors in LLM output, and make necessary corrections by reassigning missing entities and resolving cyclic dependencies. Our evaluation on real-world header file decomposition refactorings demonstrates that our method effectively overcomes the limitations of purely LLM-based techniques and outperforms the traditional state-of-the-art approach by 11%, delivering more accurate and reliable decomposition results. Our approach enables LLMs to handle lengthy header files efficiently, significantly reduces hallucinations, and ensures the reliability and practicality of the final decomposition.**

***Index Terms*—software maintenance, code refactoring, header file**

I. Introduction

Code refactoring is essential for maintaining long-lifespan software projects. As the software is enhanced, modified, and adapted to new requirements, the code becomes more complex and drifts away from its original design, thereby degrading the quality of the software [1]. Refactoring enables developers to transform poorly designed or chaotic code into well-structured systems, by strategically restructuring internal components through the relocation of methods, fields, classes, and packages. More important, Refactoring can identify and eliminate those "code smell" [2] in code, thereby maintaining code quality throughout the software life-cycle.

God Header Files [3], [4] are a newly identified type of code smell that refers to header files with large code size and wide file impact that are included by numerous other code files. Due to incremental compilation mechanisms, any modification to a God Header File necessitates recompilation of all files that include it directly or indirectly, often affecting a significant proportion of the entire software and resulting in substantial compilation time overhead. An empirical study analyzing 557 popular open-source C projects found that 37.5% of the projects were affected by God Header Files [3]. Moreover, many of these files have been modified hundreds of times during their lifespan, while each modification involves only a small proportion of the code, indicating frequent but localized changes that trigger unnecessary large-scale recompilation and underscoring the critical need for refactoring these files.

Prior work has attempted to address the God Header File problem through automated decomposition methods. For example, Wang et al. [4], [5] proposed a header file decomposition approach that utilizes several code metrics to assess entity similarity and employs a multi-view graph clustering algorithm to cluster code entities within header files, achieving significant reductions in recompilation time. However, expert-designed code similarity metrics cannot accurately and comprehensively capture the semantic essence and functional relationships of code entities. Research by Fakhoury et al. [6] demonstrates a significant gap between high metric scores and low developer acceptance. Real-world decomposition practices also illustrate this limitation. For instance, in the

* Corresponding authors.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00037

`PolarDB-for-PostgreSQL` project [1], developers decomposed `partition.h` by organizing entities around conceptual groupings—such as placing metadata management data structures together despite having lower structural or textual similarities with each other than with other entities. Such semantic-driven decomposition requires deep understanding of code functionality that goes beyond what traditional similarity metrics can reliably provide.

Effective header file decomposition requires semantic understanding capabilities that can bridge the gap between syntactic code structure and functionality. Large Language Models (LLMs) have emerged as a promising solution, offering a more holistic interpretation of code functionality compared to traditional code metrics and showing potential to provide more effective refactoring suggestions [7]–[9]. To explore LLMs' strengths and limitations in header file decomposition, we conducted a preliminary study. The findings revealed that while LLM-generated decompositions often closely resemble those produced by human developers, several critical issues emerged that hinder their practical application. First, LLMs **struggle with lengthy header files** due to token limitations and reduced effectiveness on long inputs, frequently failing to produce output or omitting numerous code entities. Second, LLMs **suffer from hallucination**, generating incomplete or spurious decomposition results due to their inability to accurately track and memorize numerous code entities and their intricate relationships within a header file. Third, LLM-generated decomposition suggestions may **contain cyclic dependencies**, where the include relationships among decomposed files form circular references. As an architectural anti-pattern [10], cyclic dependencies are particularly problematic in header files since they can cause compilation failures. These limitations significantly hinder the practical applicability of LLM-based refactoring approaches, highlighting the necessity for hybrid solutions that can address input length constraints, reduce hallucinations, and prevent cyclic dependencies.

To address these challenges, we propose *HFDecomposer*, a hybrid approach that enhances LLMs with staged grouping and dehallucination techniques to effectively decompose header files. Our approach introduces a two-stage grouping framework to handle lengthy header files: it first groups strongly related code entities using traditional similarity metrics, then presents these groups with their summarizations to the LLM for higher-level semantic aggregation. To mitigate LLM hallucinations, we enhance prompts with factual knowledge extracted from static analysis, including code entities, their dependencies and invocations. Then our approach detects hallucinations and errors in the LLM-generated decomposition, corrects them by reassigning missing entities and resolving cyclic dependencies. This process ensures a practical and reliable decomposition, after which the refactoring is automatically performed.

We evaluated *HFDecomposer* on real-world header file decomposition refactorings extracted from commits of the top 100 open-source C projects on GitHub. The results demonstrate that our approach effectively overcomes the limitations of purely LLM-based solutions by enabling LLMs to handle lengthy header files efficiently, significantly reducing hallucinations in their output, and mitigating cyclic dependency issues. This ensures reliable and practical decompositions that align with real-world developer practices. Compared to traditional state-of-the-art approaches, our method achieves an 11% improvement in decomposition accuracy.

Compared to existing work, this paper makes the following contributions:

- We introduce *HFDecomposer*, a hybrid header file decomposition approach that integrates LLMs with code analysis-based dehallucination techniques to enhance decomposition accuracy while ensuring the reliability and practical applicability of LLM-generated solutions.
- We propose a two-stage grouping framework that organizes and summarizes related code entities before presenting them to the LLM, enabling LLMs to handle lengthy code file input more effectively.
- We present a dataset of 153 real-world header file decomposition refactorings [11] and demonstrate through comprehensive experiments that *HFDecomposer* achieves 11% accuracy improvement over state-of-the-art methods while addressing key LLM limitations.

II. Preliminary Study

As LLMs are trained on extensive code repositories authored by real developers, we hypothesize that they can generate header file refactoring suggestions that better align with developer practices. In this section, we conduct a preliminary study to explore the application of LLMs in decomposing header files, focusing on both their advantages and challenges in performing this task.

A. Setup and results

To assess the capability of LLMs in decomposing header files, we constructed a dataset of real-world header file decomposition refactorings through the following systematic approach: First, we selected C projects from GitHub ranked in the top 100 by star count, ensuring high-quality codebases with substantial developer engagement and mature development practices. Second, we adapted RefactoringMiner [14], a state-of-the-art refactoring discovery tool, to identify header file decomposition refactorings from commit histories in the default branch of each project. To enable C header file analysis, we made three key modifications: (1) replaced the Java parser with Tree-sitter for C code parsing, (2) implemented coarse-grained entity-level analysis to detect movement of complete code entities (functions, data structures, macros) between files, and (3) adapted entity matching logic to handle C-specific variations. Our adapted tool analyzes commits by examining the intersection of header file contents before and after each commit, focusing on code entity movements to identify decomposition refactorings. Third, we filtered for high-quality decompositions to exclude cases involving substantial additions or changes to

[1] https://github.com/ApsaraDB/PolarDB-for-PostgreSQL/commit/da6f3e45ddb68ab3161076e120e7c32cfd46d1db

TABLE I
PRELIMINARY STUDY

Method	Failed Cases	Hallucination		Cyclic Dependency		Accuracy(%)				
		Incomplete Cases	*Avg Missing Entities*	*Cyclic Cases*	*Avg Moving Steps*	*ACC*	*F1*	*MoJoFM*	*ARI*	*NMI*
Wang et al. [4]	0	-	-	45	7.8	70.1	65.2	78.0	21.9	25.8
GPT-4o [12]	1	122	14.2	21	6.0	75.4	68.3	80.2	30.2	34.7
DeepSeek-V3 [13]	7	107	5.1	7	6.7	76.0	69.5	81.0	33.1	38.0

functionalities. Each decomposition was retained only if it met the following criteria:

- No more than 10% of code entities in the original header file were deleted during the refactoring.
- No more than 10% of code entities in each new sub-header file were newly added during the refactoring.
- Each code entity from the original header file appeared at most once across the decomposed sub-header files, preventing duplication artifacts that could compromise evaluation validity.

Our final dataset comprises 153 header file decomposition refactorings, each consisting of the original header file and the corresponding developer-implemented decomposition solution. The original header files exhibit considerable diversity, ranging from 16 to 3,162 lines of code, with an average of 403 lines. The number of code entities—defined as the atomic units for decomposition including macro definitions, type declarations, variable declarations, and function declarations—varies from 4 to 767 per file, with an average of 75.

We applied Wang et al.'s state-of-the-art header file decomposition approach [4] alongside two leading LLMs, GPT-4o [12] and DeepSeek-V3 [13], to decompose the original header files in our dataset. The quality of decomposition results was assessed by measuring their similarity to the ground-truth developer refactorings using established clustering evaluation metrics: Accuracy (ACC), F1-score, MoJoFM, Adjusted Rand Index (ARI), and Normalized Mutual Information (NMI), as detailed in Section IV-D.

Table I compares the results of Wang's approach with those of the two LLMs. Wang's approach successfully generated feasible decompositions for all header files in the dataset. Conversely, the two LLMs suffered from hallucinations, resulting in incomplete outputs or even failures. GPT-4o produced outputs for all but one header files, but struggled to generate complete results, with 122 cases of incomplete decompositions and an average of 14.3 missing entities per file. Its performance was particularly poor for lengthy header files, such as `cache.h` from the `git` project, which contains 497 code entities but was reduced to just 203 entities in GPT-4o's output. As for DeepSeek-V3, it refused to produce outputs for seven lengthy header files, and for the remaining files, it also generated incomplete results. However, it performed better than GPT-4o, with 107 incomplete cases, with 107 incomplete cases and an average of 5.1 missing entities per file.

Despite these shortcomings, the partial decompositions generated by the LLMs exhibited higher accuracy than those from Wang's approach, suggesting that LLMs align more closely with human developers' refactoring practices. Moreover, the LLM-generated decompositions had fewer cyclic dependencies (21 cases for GPT-4o, 7 for DeepSeek-V3, and 45 for Wang's approach) and required fewer steps to resolve each cyclic dependency (6.0 steps for GPT-4o, 6.7 for DeepSeek-V3, and 7.8 for Wang's). Wang's approach, which uses a clustering algorithm that treats directed dependencies as undirected, often introduced more cyclic dependencies. Although it includes a fixing algorithm, the fixing process may compromise decomposition quality. In contrast, the LLMs demonstrated a better understanding of directed dependencies, resulting in fewer cyclic dependencies and more efficient decompositions.

B. Challenges and key ideas

Our preliminary study reveals that while LLMs demonstrate promising capabilities for human-like header file decomposition, they face several critical limitations that must be addressed to ensure practical applicability.

First, LLMs encounter significant difficulties when processing **lengthy code file inputs**. Although all header files in our dataset remained within token limits, GPT-4o failed on one file while DeepSeek-V3 failed on seven files, either producing no output or terminating prematurely. This aligns with recent findings [9] demonstrating that LLMs' performance degrades as input length increases. This limitation becomes particularly concerning as real-world header files may exceed current LLM token limits entirely. To address this challenge, we propose a **two-stage grouping framework** that enables LLMs to handle lengthy files effectively. In the first stage, strongly related code entities are clustered into fine-grained groups using traditional similarity metrics, with each group then summarized to capture essential characteristics. In the second stage, the LLM performs high-level semantic aggregation using these group summaries rather than raw code, enabling global decomposition while circumventing input length limitations.

Second, LLMs suffer from **hallucinations**, producing incomplete and spurious decomposition results. As shown in Table I, both LLMs omitted essential code entities in over 100 cases, likely due to their inability to accurately distinguish and memorize numerous entities and their complex interrelationships within header files. To mitigate this issue, we propose **factual knowledge-enhanced prompts** that provide LLMs with explicit, structured information about code entities and their intricate relationships, improving code comprehension and reducing initial hallucination rates. However, since prompting alone cannot eliminate all errors, we implement a

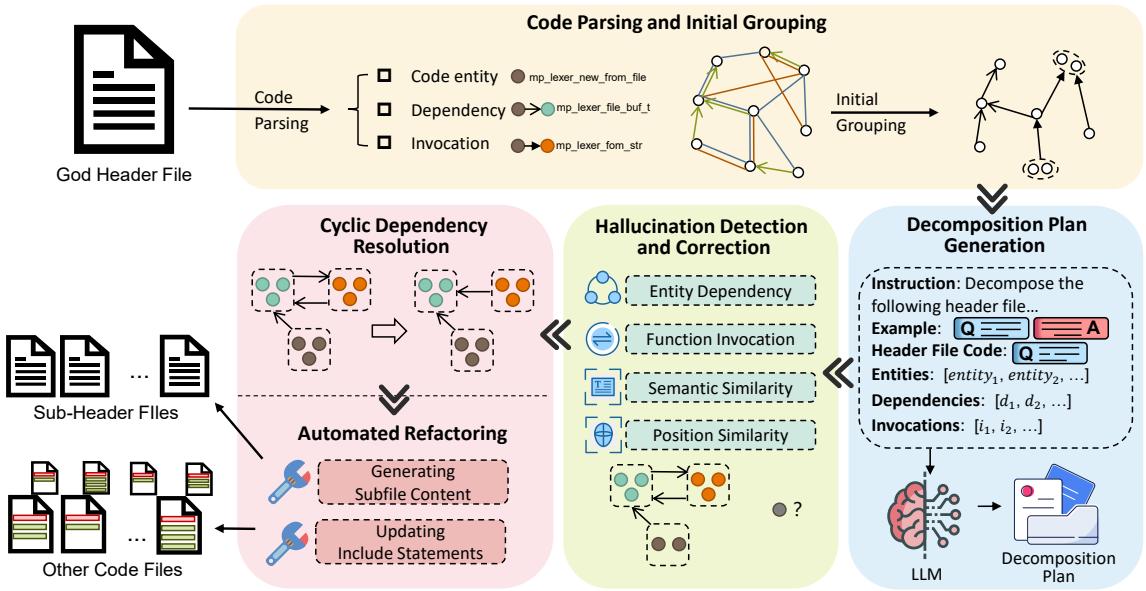

Fig. 1. Approach Overview.

hallucination detection and correction module that identifies missing entities and assigns them to appropriate sub-header files using traditional similarity metrics, ensuring a more reliable and accurate decomposition result.

Finally, LLM-generated decompositions may exhibit **cyclic dependencies**, where include relationships among decomposed files form circular references. This occurred in 21 cases with GPT-4o and 7 cases with DeepSeek-V3. Such cyclic dependencies constitute architectural anti-patterns [10] that are particularly problematic in header files, as they can cause compilation failures and violate fundamental design principles. We employ a **cyclic dependency resolution** algorithm from Wang et al.'s approach [4] to detect and eliminate these issues.

III. Approach

In this section, we present *HFDecomposer*, a hybrid approach that enhances LLMs with staged grouping and de-hallucination techniques for header file decomposition. An overview of *HFDecomposer* is illustrated in Figure 1. Our approach first parses the input header file, extracting code entities, dependencies, and invocations. For header files with lengthy content beyond LLM token capabilities, we employ staged grouping framework by initially clustering related code entities using traditional similarity metrics. Next, the original file or group summaries are provided to the LLM, with factual knowledge from static analysis incorporated in the prompt to generate a decomposition plan with reduced hallucinations. Our approach then detects hallucinations and errors in the LLM-generated plan, making corrections by reassigning missing entities and resolving cyclic dependencies, resulting in a practical decomposition plan. Finally, the approach automatically executes the refactoring according to the plan, generating new sub-header files and updating include statements in all source files that directly or transitively include the original header file.

The key steps are explained in detail in the following sections.

A. Code Parsing and Initial Grouping

HFDecomposer begins by parsing the input header file and extracting key information, including code entities and their intricate relationships. This information is utilized throughout multiple steps of the approach: initial grouping for lengthy files, constructing factual knowledge-enhanced prompts, reassigning missing entities, and resolving cyclic dependencies.

Code entities refer to definitions or declarations of macros, data types (structs, enums, unions, typedefs), variables, and functions, which cannot be further decomposed. Additionally, we extract several important code relationships between each pair of entities, each accompanied by a similarity weight computed based on code metrics from prior work. Specifically, the code relationships and their weights are defined and calculated as follows:

- **Dependency** [4]: Captures the use of a name declared by another code entity (def-use). Let $Successor_j$ denote the set of code entities using entity e_j, then

$$D_{i\to j} = \begin{cases} \frac{1}{|Successor_j|} & \text{if } e_i \text{ uses } e_j \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

$$W_d(e_i, e_j) = \max\{D_{i\to j}, D_{j\to i}\} \quad (2)$$

- **Invocation** [15]: Refers to a function entity invoking another function, typically extracted from the function body, which often resides outside the input header file. For a function e_i, let $inv(e_i, e_j)$ be the number of calls

performed by function e_i to e_j and $inv_{in}(e_j)$ be the total number of incoming calls to e_j, then,

$$I_{i\to j} = \begin{cases} \frac{inv(e_i,e_j)}{inv_{in}(e_j)} & \text{if } inv_{in}(e_j) \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

$$W_i(e_i, e_j) = max(I_{i\to j}, I_{j\to i}) \quad (4)$$

- **Semantics** [15]: Represents textual similarity between two code entities e_i and e_j. We use the Latent Semantic Indexing (LSI) [16] technique to obtain the semantic vector of an entity ($\vec{m_i}$ and $\vec{m_j}$). We then compute the semantics weight by calculating the cosine similarity of their corresponding vectors:

$$W_s(e_i, e_j) = \frac{\vec{m_i} \cdot \vec{m_j}}{|\vec{m_i}| \cdot |\vec{m_j}|} \quad (5)$$

- **Position** [17]: Measures the relative positional distance between two code entities e_i and e_j. Let p_i denote the relative position (index) of e_i, then,

$$W_p(e_i, e_j) = \frac{1}{|p_i - p_j|} \quad (6)$$

For lengthy header files exceeding the capacity of LLMs, *HFDecomposer* employs a two-stage grouping framework. This process begins by clustering code entities into fine-grained groups, which are then provided to the LLM for higher-level aggregation. In the initial grouping stage, we compute the average weights of Dependency, Invocation, and Semantics relationships and apply the efficient Louvain algorithm [18] to cluster strongly related code entities. Accuracy at this stage is crucial, as errors in the initial grouping can propagate through subsequent steps. To minimize such risks, we follow the approach of Chen et al. [17], leveraging Position relationship to split inconsecutive clusters into smaller, consecutive ones, as their study indicates that consecutive clusters are less prone to errors. Once the groups are formed, we prompt the LLM to generate concise summaries for each group. These summaries, which are substantially more compact than the original file content, serve as a compressed representation that replaces the lengthy header file in the subsequent LLM-based decomposition phase, thereby ensuring compatibility with LLM input constraints.

B. Factual Knowledge-Enhanced Prompt Construction

As LLMs struggle to distinguish and memorize a large number of code entities in a header file, we construct a factual knowledge-enhanced prompt to provide clear and structured code fact knowledge. By explicitly listing code entity names and their intricate relationships, this prompt improves the LLM's understanding of the code content and helps mitigate hallucinations.

Specifically, a factual knowledge-enhanced prompt is composed of four key components: instructions, a decomposition example, the code file content, and factual knowledge. **Instructions**: This part outlines the task of decomposing a header file into smaller, more manageable sub-header files and specifies the expected output format. **Decomposition Example**: This component provides an example of a header file along with a suggested decomposition, illustrating how the process is to be carried out. **Code File Content**: This part includes the header file name, the expected number of subfiles, and the full code content of the header file. If the file is too lengthy, the code content is replaced by the summarization list generated in the previous phase. **Factual Knowledge**: This part explicitly lists all code entities in the file, along with their Dependency and Invocation relationships, to mitigate LLM hallucinations. By providing precise and complete information, it prevents the LLM from omitting listed entities or introducing unintended ones. This enhances the quality of the LLM's suggestions, resulting in a more accurate and reliable decomposition output.

C. Hallucination Detection and Correction

Since hallucination is an inherent limitation of LLMs and cannot be fully eliminated, we employ a hallucination detection and correction module to identify and rectify errors in the LLM-generated decompositions. This module addresses two primary types of errors: 1) spurious code entities, where code entities included in the decomposition do not belong to the original header file. 2) missing code entities, which occur when certain code entities from the original header file are absent from all newly generated sub-header files.

For spurious entities, we simply remove them from the decomposition. For each missing entity, we calculate their similarity with each sub-header file in the decomposition and assign each missing entity to the sub-header file with the highest similarity score. The similarity calculation leverages the code relationship weights described in Section III-A. For a missing entity e_m and a sub-header file H_i containing entities $\{e_1, \cdots, e_k\}$, we first compute the overall similarity between e_m and each entity e_j in H_i as the average of all four relationship weights, denoted as $OverallSim(e_m, e_j)$. The similarity between missing entity e_m and sub-header file H_i is then calculated as:

$$Sim(e_m, H_i) = \frac{1}{|H_i|} \sum_{j=1}^{k} OverallSim(e_m, e_j) \quad (7)$$

Finally, the missing entity e_m is assigned to the new sub-header file with the highest similarity score.

D. Cyclic Dependency Resolution

The decomposition plan from the previous phase produces several sub-header files with potential inclusion relationships. When a code entity in one sub-header file depends on an entity in another sub-header file, an include relationship is established between them. Cyclic dependencies among these inclusion relationships render the decomposition result unacceptable and can cause compilation errors.

To fix cyclic dependencies, we employ the cyclic dependency resolution algorithm from Wang et al. [4], which extends from a two-node cycle fixing method originally proposed by Herrmann et al. [19]. For a two-node cycle involving sub-header files H_i and H_j, we can eliminate the cycle by removing the dependency edge in either direction. This can be

achieved through four possible actions: moving all ancestors or descendants of entities from H_i to H_j, or vice versa. We calculate a moving gain for each option based on the relationship weights between the moved entities and their respective sub-header files, then select the option with the optimal gain. For cycles longer than two nodes, we reduce the problem to a two-node cycle by selecting one sub-header file from the cycle and treating all remaining files as a single consolidated unit. We then evaluate all possible choices and select the best solution based on the moving gain criterion. By iteratively addressing the longest cycle first, the algorithm ensures that no cycles of equal or greater length are introduced during the resolution process, ultimately eliminating all cyclic dependencies from the decomposition result.

E. Automated Refactoring Execution

Once the user approves a decomposition plan, *HFDecomposer* automatically executes the refactorings in two steps:

1) Sub-Header File Content Generation: HFDecomposer copies the content of each code entity to its designated sub-header file based on the decomposition plan. For code entities within conditional compilation blocks (e.g., `#ifdef`), the system maintains environment stacks for both the original file and each sub-header file to ensure consistent preprocessing contexts when copying entities.

2) Include Relationship Modification: Our approach then updates include statements in all files that directly or transitively include the original header file. Each dependent file is analyzed to identify the required sub-header files, and the original include statement is replaced with appropriate includes for the necessary sub-header files. For files that transitively depend on the original header file through intermediary headers, we directly add the necessary include statements to these files to prevent compilation errors when intermediary headers no longer provide the required dependencies. This process follows Google's "Include What You Use" guideline[2].

IV. Experimental Setting

To assess the performance of our approach, we conducted evaluations on a real-world header file decomposition dataset and compared our approach with state-of-the-art methods.

A. Research Questions

To evaluate our approach, we seek to answer the following research questions:

RQ1. Can *HFDecomposer* address the limitations of LLMs in header file decomposition and outperform existing approaches? We assess the effectiveness of *HFDecomposer* in handling lengthy header file inputs, mitigating hallucinations, resolving cyclic dependencies, and achieving superior decomposition accuracy compared to baseline methods.

RQ2. How effective is our Two-stage Grouping Framework for lengthy header files? This research question evaluates both the accuracy of the initial grouping stage and the final decomposition quality when processing header files that exceed LLMs' capability.

RQ3. How does each component of our approach contribute to overall performance? We perform an ablation study to investigate the contribution of each key component, including different types of factual knowledge incorporation, the hallucination detection and correction module, and the cyclic dependency resolution module.

[2]https://google.github.io/styleguide/cppguide.html#Include_What_You_Use

TABLE II
Dataset Statistics

	Count		Mean	Min	Median	Max
# ce $<$ 200	141	# ce	53	4	40	192
		LOC	354	16	262	1621
# ce $\geq$ 200	12	# ce	343	200	279	767
		LOC	1182	359	978	3162
Total	153	# ce	75	4	43	767
		LOC	419	16	285	3162

ce: number of code entities; LOC: lines of code

B. Datasets

To evaluate our approach, we constructed a dataset comprising 153 real-world header file decomposition refactorings, as described in Section II-A. Based on findings from our preliminary study indicating that LLMs exhibit degraded performance on header files containing 200 or more code entities, we partitioned the dataset into two groups based on entity count. Table II presents the statistical characteristics of each group and the complete dataset. For header files with fewer than 200 code entities, our approach directly provides their complete code content to the LLM for decomposition. For files containing 200 or more entities, we employ the two-stage grouping framework, first clustering and summarizing related code entities before presenting the condensed representation to the LLM.

C. Baselines

We compare our approach against three baseline methods to evaluate its effectiveness:

Wang et al. [4]: This represents the current state-of-the-art approach for header file decomposition. It constructs a code entity graph using multiple similarity metrics and applies a multi-view graph clustering algorithm to partition code entities into coherent sub-header files.

GPT-4o [12]: We evaluate the raw performance of GPT-4o, one of the most advanced LLMs, applied directly to header file decomposition without our enhancement techniques. This baseline helps assess the inherent capabilities and limitations of state-of-the-art LLMs on this task.

DeepSeek-V3 [13]: We also include DeepSeek-V3, a leading open-source LLM, to evaluate performance across different models. This provides insight into how our approach performs with various LLM backends.

For the LLM-based baselines, we provide the same decomposition instructions and examples used in our approach, but without the factual knowledge enhancement, staged grouping, or error correction mechanisms. This ensures a fair comparison that isolates the contribution of our specific enhancements.

TABLE III
OVERALL PERFORMANCE

Method	Factual Knowledge	Failed Cases	Hallucination		Cyclic Dependency		Accuracy(%)				
			Incomplete Cases	*Avg Completing Steps*	*Cyclic Cases*	*Avg Moving Steps*	*ACC*	*F1*	*MoJoFM*	*ARI*	*NMI*
Wang et al. [4]	-	0	-	-	45	7.8	70.1	65.2	78.0	21.9	25.8
Gpt-4o [12]	-	1	122	14.2	21	6.0	75.4	68.3	80.2	30.2	34.7
Ours(Gpt-4o)	Entity	0	10	2.9	13	6.1	**76.0**	**70.8**	**81.8**	**34.2**	**39.3**
	Entity + Dependency	0	8	13.6	10	2.8	74.0	67.3	80.0	28.2	33.9
	Entity + Invocation	0	6	2.8	17	7.0	74.6	67.7	80.5	30.5	35.7
	Entity + Both	0	7	4.6	13	2.0	74.3	67.5	79.8	28.4	33.9
Deepseek-V3 [13]	-	7	107	5.1	7	6.7	76.0	69.5	81.0	33.1	38.0
Ours(Deepseek-V3)	Entity	0	2	2.0	10	5.1	75.4	70.1	80.9	32.3	37.3
	Entity + Dependency	0	4	1.5	4	1.8	**78.0**	**71.7**	**82.9**	**37.8**	**42.7**
	Entity + Invocation	0	2	1.0	6	6.7	76.4	70.9	81.7	35.1	39.5
	Entity + Both	0	1	1.0	3	3.0	76.0	69.9	81.9	34.6	39.4

D. Evaluation Metrics

To assess the accuracy of our approach, we measured each generated decomposition result based on its closeness to the expected result using a set of common metrics, which are described below.

- **Accuracy (ACC)** [20] measures the proportion of correctly predicted cluster labels by building a mapping between predicted cluster labels and ground truth cluster labels through the Kuhn-Munkres [21] algorithm.
- **F1-score (F1)** is also a metric for evaluating the clustering performance, which combines precision and recall.
- **MoJoFM** [22] quantifies the number of Move and Join operations required to transform architecture A into B. It is calculated as:

$$MoJoFM(A,B) = (1 - \frac{mno(A,B)}{\max(mno(\forall A,B))}) \times 100\% \quad (8)$$

where $mno(A,B)$ is the minimum number of *Move* or *Join* operations needed to transform the partition A to B. MoJoFM scores range from 0% to 100%, where a higher value indicates a higher similarity between two partitions.
- **Normalized Mutual Information (NMI)** [23] measures the mutual information between the predicted clusters and ground truth clusters. It normalizes the mutual information to fall within a range of 0 to 1, with 1 indicating perfect clustering agreement.
- **Adjust Rand Index (ARI)** [24] is based on the pairwise similarity between predicted labels and ground truth labels. It adjusts for random chance, providing a normalized score between -1 and 1, where higher values indicate better clustering.

V. EXPERIMENTAL RESULTS

We now describe our evaluation results for each research question.

A. RQ1. Overall Performance

To answer RQ1, we evaluated our approach against three baselines: Wang's approach, vanilla GPT-4o and DeepSeek-V3. For header files containing 200 or more code entities, we employed the two-stage grouping framework, as our preliminary study demonstrated that both LLMs exhibit degraded performance on such lengthy inputs. Table III presents the evaluation results, including the number of cases with generation failures, hallucinations, or cyclic dependency issues, alongside the average accuracy of the final decomposition results.

From Table III, we make the following observations:

- **Enhanced processing of lengthy header files**. Our approach effectively addresses LLM limitations when processing lengthy header file inputs. Vanilla GPT-4o and DeepSeek-V3 failed to generate output for 1 and 7 header files, respectively, due to degraded performance on lengthy input. With our two-stage grouping framework, both models successfully processed all header files in the dataset, achieving 100% completion rate and significantly improving their generalizability.
- **Mitigation of hallucination**. Our approach substantially reduces hallucination in LLM-generated outputs and ensures reliable final decomposition results. The number of incomplete cases dropped dramatically from 122 to below 10 for GPT-4o and from 107 to below 5 for DeepSeek-V3. Additionally, the average number of missing entities also decreased. This improvement stems from our factual knowledge-enhanced prompts and two-stage grouping framework. Explicitly listing entity names and relationships within the header file guides the LLM's attention and enhances code understanding. For lengthy inputs where LLMs typically omit numerous entities, initial grouping reduces cognitive load and enables higher-quality outputs. Furthermore, our hallucination detection and correction module serves as the final safeguard, completely eliminating all hallucinations in the final decomposition results across all test cases.
- **Reduction of cyclic dependency issues**. Our approach significantly enhances LLMs' ability to generate decompositions with fewer cyclic dependencies. While vanilla LLMs already produce fewer cyclic cases than Wang's approach, our factual knowledge-enhanced prompts further reduce these issues. By explicitly providing dependencies among code entities, LLMs gain better architectural awareness, generating output with minimal post-processing adjustments.

TABLE IV
EFFECTIVENESS OF TWO-STAGE GROUPING FRAMEWORK

Header File	Project	# Code Entities	# Generated Groups	# Accurate Groups	Accuracy (%)	Wang et al.			Ours(DeepSeek-V3)		
						ACC	*F1*	*MoJoFM*	*ACC*	*F1*	*MoJoFM*
ui.h	goaccess	200	53	51	96.2	61.0	61.0	60.6	90.5	90.5	90.4
tdtaformat.h	TDengine	211	49	48	98.0	77.0	60.9	86.5	56.3	36.0	89.1
shared.h	hashcat	214	39	39	100.0	58.6	50.7	89.2	63.1	38.7	89.7
soc.h	esp-idf	244	13	13	100.0	51.2	48.4	85.5	51.6	48.7	85.5
awk.h	cosmopolitan	262	46	45	98.0	61.3	55.2	86.1	70.1	41.2	84.4
quantum_keycodes.h	qmk_firmware	271	23	22	95.7	63.5	58.9	84.4	78.1	78.1	89.0
git-compat-util.h	git	287	130	130	100.0	53.5	41.1	89.5	63.6	55.2	89.4
taosmsg.h	TDengine	312	45	44	97.8	91.3	91.0	91.3	50.3	47.8	63.2
cache.h	git	382	124	121	97.6	58.0	48.4	88.3	63.1	55.8	88.5
cache.h	git	480	145	136	93.8	64.0	59.5	77.8	77.9	43.8	77.8
avcodec.h	FFmpeg	488	9	9	100.0	72.4	71.7	72.3	68.5	67.9	68.4
cache.h	git	767	254	253	99.6	75.9	67.1	87.7	76.9	68.0	87.8
Average		343	77.5	75.9	98.1	65.6	59.5	81.2	67.5	56.0	82.4

- **Higher decomposition accuracy**. Our approach consistently improves upon vanilla LLM performance and achieves superior decomposition accuracy across all evaluation metrics compared to Wang's state-of-the-art method. The best-performing configuration, DeepSeek-V3 with entity and dependency knowledge, attains 78.0% ACC, representing an 11% improvement over Wang's approach (70.1%). These results demonstrate that our hybrid approach produces decompositions that significantly better align with real-world developer refactoring practices.

To answer RQ1, our approach effectively addresses LLM limitations in header file decomposition while leveraging their code understanding capabilities. By integrating staged grouping and dehallucination strategies, our approach achieves 100% success on lengthy files, over 90% reduction in hallucinations, fewer cyclic dependencies, and 11% accuracy improvement over the state-of-the-art method.

B. RQ2. Effectiveness of Two-stage Grouping Framework

For lengthy header file inputs, our approach employs a two-stage grouping framework to enable effective LLM processing. This framework first groups strongly related code entities using traditional similarity metrics, then provides summaries of each group to the LLM for higher-level decomposition. To validate its effectiveness, we evaluated both the initial grouping quality and final decomposition accuracy for 12 lengthy header files containing 200 or more code entities. Table IV presents the detailed results of this evaluation.

We first evaluated the quality of initial grouping results. Table IV shows the number of code entities in each header file, the number of groups generated after initial grouping, and the number and percentage of accurate groups preserved in the ground truth decomposition. Our initial grouping achieves substantial input reduction while maintaining high accuracy, condensing an average of 343 code entities into 77.5 groups with 98.1% accuracy. The reduction ratio varies across files due to our consecutive constraint, which splits groups containing non-contiguous code entities. Although this constraint limits the reduction ratio, it significantly enhances grouping accuracy—a critical factor since errors in initial grouping can propagate through subsequent processing stages.

We also compared the final decomposition results between Wang's traditional approach and our method using DeepSeek-V3. While vanilla DeepSeek-V3 failed to generate valid outputs for 7 out of 12 lengthy header files, our two-stage grouping framework enabled successful processing of all files. The final decomposition accuracy was comparable to Wang's approach, with our method achieving slight improvements in ACC and MoJoFM scores, though with a marginal decrease in F1 score. However, when compared to performance on shorter files in the overall dataset, the decomposition quality for lengthy header files remains lower, indicating that even with our framework, LLMs continue to face challenges with complex, lengthy inputs.

To answer RQ2, our two-stage grouping framework generates highly accurate initial groups, enables LLMs to handle lengthy inputs previously unmanageable, and achieves comparable decomposition accuracy to traditional approaches. Nonetheless, output quality for lengthy files remains lower, indicating opportunities for further improvement.

C. RQ3. Ablation Study

To evaluate the contribution of each component in our approach, we conducted an ablation study examining: (1) the impact of different types of factual knowledge in prompts, (2) the effectiveness of hallucination detection and correction, and (3) the benefits of cyclic dependency resolution. Table V presents the comprehensive results across both LLM models. In this table, "adjustment" refers to the post-processing of LLM-generated outputs, including Hallucination Detection and Correction (Section III-C) and Cyclic Dependency Resolution (Section III-D). The "before adjustment" columns present accuracy scores of raw LLM outputs, while "after adjustment" columns show the final results of our approach.

Our factual knowledge-enhanced prompts incorporate three types of information: code entities, dependency, and invocation. Each knowledge type contributes differently to decomposition quality. For the entity knowledge, it significantly reduces hallucination in both LLMs, minimizing omitted entities

TABLE V
ABLATION STUDY

Method	Factual Knowledge	Before Adjustment			Hallucination		Cyclic Dependency		After Adjustment		
		ACC	*F1*	*MoJoFM*	*Incomplete Cases*	*Avg Completing Steps*	*Cyclic Cases*	*Avg Moving Steps*	*ACC*	*F1*	*MoJoFM*
Ours(Gpt-4o)	Entity	75.9	70.8	81.8	10	2.9	13	6.1	76.0	70.8	81.8
	Entity + Dependency	73.9	67.2	80.0	8	13.6	10	2.8	74.0	67.3	80.0
	Entity + Invocation	74.5	67.8	80.6	6	2.8	17	7.0	74.6	67.7	80.5
	Entity + Both	74.4	67.6	79.8	7	4.6	13	2.0	74.3	67.5	79.8
Ours(Deepseek-V3)	Entity	75.2	70.0	81.0	2	2.0	10	5.1	75.4	70.1	80.9
	Entity + Dependency	77.9	71.7	82.9	4	1.5	4	1.8	78.0	71.7	82.9
	Entity + Invocation	76.4	71.0	81.8	2	1.0	6	6.7	76.4	70.9	81.7
	Entity + Both	75.9	69.9	81.9	1	1.0	3	3.0	76.0	69.9	81.9

in their outputs and thereby improving final accuracy. The knowledge of dependency notably reduces cyclic dependency issues, as evidenced by fewer cyclic cases and fewer required fixes for both LLMs.This effectiveness stems from dependency knowledge directly informing the structural relationships used to construct include relationships among sub-header files. By emphasizing dependencies in the header file input, it compensates for LLMs' inability to fully capture or recall such information. In contrast, the invocation knowledge has minimal or negative impact on performance. This occurs because invocation knowledge is extracted from implementation files rather than the header file being decomposed, representing functional relationships which are not directly relevant to structural dependencies. The impact of dependency knowledge and invocation knowledge on accuracy varies between the two LLMs. DeepSeek-V3 benefits from additional knowledge, showing improved performance, while GPT-4o experiences a decline in accuracy. This discrepancy can be attributed to differences in the LLMs' ability to process and utilize relational knowledge effectively. Additionally, the inclusion of more knowledge increases the input length, which heightens the cognitive burden on the LLMs and may adversely affect their performance.

The adjustment processes, including hallucination correction and cyclic dependency resolution, enhance reliability while preserving accuracy. In most configurations, these post-processing steps eliminate hallucinations and resolve architectural issues without compromising decomposition quality, with some configurations showing slight accuracy improvements.

To illustrate the impact of factual knowledge, we present an example using the DeepSeek-V3 to decompose the `threads.h` file from the `vlc` project. In the developer-conducted, this header file was split into two sub-header files: `threads.h`, containing macros and type declarations, and `threads_func.h`, containing functions. Figure 2 shows the decomposition results when only entity knowledge is included in the prompt. The LLM produces an almost perfect result, except for the placement of the code entity `vlc_thread_wrapper`. While this entity has four dependencies in both sub-header files, the LLM incorrectly assigns it to `threads.h`. This error likely stems from semantic confusion due to overlapping subwords, such as `wrapper_t`, causing the LLM to prioritize textual similarity

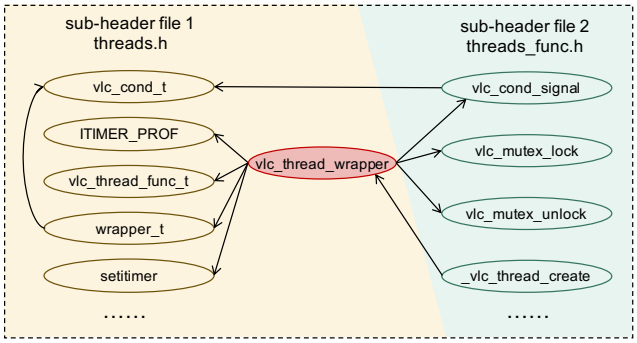

Fig. 2. DeepSeek-V3 decomposition of `threads.h` without dependency knowledge

over structural dependencies. Consequently, this misplacement introduces a cyclic dependency, as several functions in `threads_func.h` rely on declarations from `threads.h`. When dependency knowledge is incorporated into the prompt, the explicit specification of dependency relationships and their directions enables the LLM to correctly interpret structural constraints. This results in accurate entity placement and eliminates the cyclic dependency, producing a decomposition that perfectly matches the developer's solution.

To answer RQ3, each component contributes meaningfully to overall performance: entity knowledge reduces hallucination, dependency knowledge mitigates cyclic dependencies, and adjustment processes enhance reliability without compromising accuracy.

VI. DISCUSSION

In this section, we discuss the limitations and threats to validity of our approach.

A. Limitations and Future Work

When applying *HFDecomposer* in practice, certain limitations and challenges remain that warrant further improvement in the future:

1) Customized requirements: Software refactoring is both a science and an art, often influenced by the subjective interpretations of developers [8]. Research on software modularization [25] has shown that developers may hold differing

opinions on the same refactoring task due to varying perspectives, such as logical, process, development, and physical viewpoints [26]. Although our current approach does not support fully customized requirements, developers can tailor the prompts to align the decomposition with their preferred viewpoints. In the future, we aim to explore the ability of LLMs to adapt to customized requirements and to automatically infer potential requirements by analyzing development history.

2) Economic and time cost: Our experiments invoke LLMs through commercial APIs, which require payment for API tokens—-an important consideration for practical use. Additionally, LLM-based approaches often face challenges with response times, which must also be carefully considered.

B. Threats to Validity

Our experiment results may suffer from several threats to validity. We discuss internal validity and external validity respectively.

1) Internal Validity: The threats to internal validity include: a) Non-determinism of LLMs. LLMs exhibit non-deterministic behavior, leading to variability in generated refactoring suggestions and impacting the reproducibility and consistency of our results. To mitigate this, we set the temperature parameter to 0 for more stable predictions. b) Data leakage. The risk of data leakage arises if LLMs were pre-trained on projects similar to those in the evaluation dataset, potentially inflating performance metrics. Our evaluation dataset were mined from intermediate versions, minimizing overlap with pre-training data. c) Non-uniqueness of golden set. Our approach was evaluated using a golden dataset comprising real-world header file decompositions performed by developers, with accuracy metrics computed based on it. However, decomposition is often subjective, and alternative solutions differing from the golden set can also be valid.

2) External Validity: The evaluation dataset was mined from top 100 C projects from Github, selected to ensure diversity in scale and coverage across various business domains, thereby addressing concerns about the generalizability of results. However, the dataset is limited to open-source projects written in C language, which constrains the applicability of our conclusions to other programming languages or non-open-source environments, even though our approach is conceptually language-agnostic. Additionally, the selection of highly starred projects, which are typically of higher quality, may introduce bias by underrepresenting less popular or lower-quality projects.

Another potential threat to generalizability lies in the use of specific LLMs. To address this, we conducted experiments on two different LLMs. However, both models represent state-of-the-art systems, so our findings may not generalize to smaller or less capable LLMs. As LLM technology continues to evolve, particularly with upcoming iterations trained on larger and more diverse datasets, we anticipate that these advancements will lead to further improvements in the outcomes of our approach.

VII. Related Work

Given the relevance of the tasks and the techniques, we present the related work about code file decomposition and LLM-based software refactoring.

A. Code File Decomposition

Numerous studies have focused on refactoring large code files through decomposition to enhance system readability and maintainability. Among these, the God Class is a widely studied concept, representing complex classes that centralize a system's functionality [27]. While some researchers debate the necessity of refactoring God Classes [28], [29], many have proposed semi-automatic [30] or automatic approaches, often relying on code metrics to quantify the similarity between software entities and applying clustering algorithms to produce new classes. Various code metrics have been developed for this purpose. De Lucia et al. [31] incorporated structural and semantic similarities, while Fokaefs et al. [32] utilized Jaccard distance between sets of software entities. Bavota et al. [15], [33] introduced metrics based on method invocations and employed Latent Semantic Indexing (LSI) [16] to capture semantic relationships. Further advancements explored richer semantic metrics, including those based on Latent Dirichlet Allocation [34] and document similarity [35]. Recently, ClassSplitter [17] observed strong semantic correlations among physically adjacent entities and refined clustering by integrating traditional code metrics with the relative physical positions of entities. Decomposition of header files has also gained attention. Wang et al. [4] pioneered the decomposition of God Header Files, proposing a multi-view graph clustering approach to generate decomposition suggestions and a heuristic algorithm to resolve cyclic dependencies in the results.

These approaches provide reliable decomposition with broad applicability, often improving software quality metrics. However, manually designed code similarity metrics fall short of fully capturing the semantic and functional relationships of code entities. By integrating traditional methods with LLMs, our approach achieves greater accuracy while maintaining reliability.

B. LLM-based Software Refactoring

Recently, LLMs have gained attention for their ability to understand code and address issues, driving research on their use in software refactoring, code smell elimination, and quality improvement [36], [37]. Several empirical studies have explored the capabilities of LLMs in refactoring. Liu et al. [9] demonstrated that LLMs can identify refactoring opportunities, particularly those with critical severity [38], and propose high-quality solutions, though they may occasionally introduce risky changes. Similarly, Cordeiro et al. [7] and DePalma et al. [39] found that while LLMs perform well in addressing systematic and repetitive issues, they struggle with complex, context-dependent code smells compared to human developers. LLMs have also been integrated into innovative refactoring approaches. Shirafuji et al. [40] and Choi et al. [41] utilized LLMs to reduce method complexity, achieving significant

reductions in average code lines and cyclomatic complexity through iterative refinement or multiple candidate outputs. Pomian et al. [8], [42] targeted long methods, using LLMs to generate numerous Extract Method solutions, filtering out invalid ones, and refining and reranking the remainder to select the optimal solution. Batole et al. [43] proposed a multi-agent framework leveraging LLMs for identifying code design issues, employing natural language summaries, context-aware prompts, and LLM-based ranking for results. Wu et al. [44] introduced iSMELL, an ensemble method combining multiple code smell detection tools within a Mixture of Experts (MoE) architecture, enhancing LLM performance by integrating detection results to guide refactoring.

The practical applicability of LLM-based solutions is limited by token constraints and inherent hallucinations. Our approach overcomes these challenges through a staged grouping framework, factual knowledge-enhanced prompts, and mechanisms for detecting and correcting hallucinations.

VIII. Conclusion

This paper introduces *HFDecomposer*, a hybrid approach that enhances LLMs with staged grouping and dehallucination techniques for effective header file decomposition. By integrating a two-stage grouping framework, knowledge-enhanced prompts, hallucination detection and correction, and cyclic dependency resolution, our method addresses the limitations of purely LLM-based solutions. It enables LLMs to efficiently process lengthy header files, minimizes hallucinations, and mitigates cyclic dependency issues. Comprehensive evaluations on 153 real-world header file decompositions demonstrate that *HFDecomposer* significantly outperforms state-of-the-art methods, delivering more accurate and reliable results.

In the future, we aim to enhance our approach to support customized decomposition requirements, as developers often hold differing opinions due to varying perspectives such as logical, process, development, and physical viewpoints. While developers can currently tailor prompts to their preferred viewpoints, we plan to explore enabling LLMs to automatically adapt to specific developer preferences and infer decomposition requirements by analyzing project development history.

Acknowledgment

This work is supported by the National Key Research and Development Program of China under Grant No.2023YFB4503803.

References

[1] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[2] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[3] Y. Wang, W. Chang, Y. Zou, and B. Xie, "An exploratory study on god header files in open-source c projects," in *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, 2024, pp. 477–486.

[4] Y. Wang, W. Chang, T. Deng, Y. Zou, and B. Xie, "Decomposing god header file via multi-view graph clustering," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2024, pp. 112–124.

[5] Y. Wang, J. Sun, T. Deng, W. Chang, Y. Zou, and B. Xie, "Headersplit: An automated tool for splitting header files in c projects," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, 2025, pp. 1124–1128.

[6] S. Fakhoury, D. Roy, A. Hassan, and V. Arnaoudova, "Improving source code readability: Theory and practice," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 2–12.

[7] J. Cordeiro, S. Noei, and Y. Zou, "An empirical study on the code refactoring capability of large language models," 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:273821330

[8] 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 *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2024, pp. 275–287.

[9] 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," *Automated Software Engineering*, vol. 32, no. 1, p. 26, 2025.

[10] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE software*, vol. 35, no. 3, pp. 56–62, 2018.

[11] HFDecomposer, "Hfdecomposer," https://github.com/HFDecomposer/HFDecomposer, 2025.

[12] OpenAI, "Hello gpt-4o," https://openai.com/index/hello-gpt-4o/, 2024, accessed: 12/2024.

[13] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[14] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2022.

[15] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: An improved method and its evaluation," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1617–1664, 2014.

[16] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[17] T. Chen, Y. Jiang, F. Fan, B. Liu, and H. Liu, "A position-aware approach to decomposing god classes," in *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024, pp. 129–140.

[18] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.

[19] J. Herrmann, M. Y. Ozkaya, B. Uçar, K. Kaya, and Ü. V. çatalyürek, "Multilevel algorithms for acyclic partitioning of directed acyclic graphs," *SIAM Journal on Scientific Computing*, vol. 41, no. 4, pp. A2117–A2145, 2019.

[20] A. Lutov, M. Khayati, and P. Cudré-Mauroux, "Accuracy evaluation of overlapping and multi-resolution clustering algorithms on large datasets," in *2019 IEEE International Conference on Big Data and Smart Computing (BigComp)*. IEEE, 2019, pp. 1–8.

[21] L. Lovász and M. D. Plummer, *Matching theory*. American Mathematical Soc., 2009, vol. 367.

[22] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* IEEE, 2004, pp. 194–203.

[23] A. F. McDaid, D. Greene, and N. Hurley, "Normalized mutual information to evaluate overlapping community finding algorithms," *arXiv preprint arXiv:1110.2515*, 2011.

[24] K. Y. Yeung and W. L. Ruzzo, "Details of the adjusted rand index and clustering algorithms, supplement to the paper an empirical study on principal component analysis for clustering gene expression data," *Bioinformatics*, vol. 17, no. 9, pp. 763–774, 2001.

[25] F. Meng, Y. Wang, C. Y. Chong, H. Yu, and Z. Zhu, "Evolution-aware constraint derivation approach for software remodularization," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–43, 2024.

[26] P. B. Kruchten, "The 4+ 1 view model of architecture," *IEEE software*, vol. 12, no. 6, pp. 42–50, 1995.

[27] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[28] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *2010 IEEE international conference on software maintenance*. IEEE, 2010, pp. 1–10.

[29] R. Pérez-Castillo and M. Piattini, "Analyzing the harmful effect of god class refactoring on power consumption," *IEEE software*, vol. 31, no. 3, pp. 48–54, 2014.

[30] N. Anquetil, A. Etien, G. Andreo, and S. Ducasse, "Decomposing god classes at siemens," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 169–180.

[31] A. De Lucia, R. Oliveto, and M. Vorraro, "Using structural and semantic metrics to improve class cohesion," in *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 27–36.

[32] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander, "Decomposing object-oriented class modules using an agglomerative clustering technique," in *2009 IEEE International Conference on Software Maintenance*. IEEE, 2009, pp. 93–101.

[33] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying extract class refactoring opportunities using structural and semantic cohesion measures," *Journal of Systems and Software*, vol. 84, no. 3, pp. 397–414, 2011.

[34] P. Saha Akash, A. Z. Sadiq, and A. Kabir, "An approach of extracting god class exploiting both structural and semantic similarity," in *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*. INSTICC, SciTePress, 2019, pp. 427–433.

[35] T. Jeba, T. Mahmud, P. S. Akash, and N. Nahar, "God class refactoring recommendation and extraction using context based grouping," 2020.

[36] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. C. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ArXiv*, vol. abs/2308.10620, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:261048648

[37] Y. Chen, J. Wu, X. Ling, C. Li, Z. Rui, T. Luo, and Y. Wu, "When large language models confront repository-level automatic program repair: How well they done?" *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 459–471, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:268201602

[38] L. L. Silva, J. R. D. Silva, J. E. Montandon, M. Andrade, and M. T. Valente, "Detecting code smells using chatgpt: Initial insights."

[39] 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*, vol. 249, p. 123602, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417424004676

[40] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe, "Refactoring programs using large language models with few-shot examples," *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 151–160, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:265295529

[41] J. Choi, G. An, and S. Yoo, "Iterative refactoring of real-world open-source programs with large language models," in *International Symposium on Search Based Software Engineering*. Springer, 2024, pp. 49–55.

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

[43] F. Batole, D. OBrien, T. N. Nguyen, R. Dyer, and H. Rajan, "An llm-based agent-oriented approach for automated code design issue localization," in *ICSE'2025: The 47th International Conference on Software Engineering*, April 27-May 3 2025.

[44] 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," *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1345–1357, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:273476633