Let the Comments Speak: A Multi-Agent Framework based on Large Language Model for Comment-Guided Code Refactoring

Zixuan Wang* , **Wutong Yu*** , **Deyu Zhou†**
School of Computer Science and Engineering,
Key Laboratory of Computer Network and Information Integration,
Ministry of Education, Southeast University, China
{220242302, 230249001, d.zhou}@seu.edu.cn

Abstract

Code refactoring is essential for software maintainability, yet current Large Language Model (LLM) based frameworks primarily focus on syntax and neglect the vital semantic signals in code comments. As pointed out in Fowler's refactoring theory, explanatory comments function as semantic anchors that provide necessary guidance for method extraction. Moreover, research reports show that more than 84 percent of codebases lack appropriate code comments, which hinders the leverage of such guidance. To bridge this gap, we propose MACOR, a Multi-Agent framework for COmment-guided code Refactoring. In specific, MACOR populates original code with precise comments to provide necessary semantic guidance for the subsequent refactoring process. These generated signals are employed to retrieve expert examples. An iterative feedback is incorporated loop for validation. Experiments are conducted on three benchmarks using three base LLMs. Experimental results show that MACOR significantly optimizes code quality and achieves higher developer acceptance compared to the representative baselines.

1 Introduction

"A heuristic we follow is that whenever we feel the need to comment something, we write a method instead."

— MARTIN FOWLER

Refactoring is the process of optimizing the internal design and structure of existing code without changing its external behavior (Fowler, 1999). This practice is essential for managing technical debt and ensuring that systems adapt to evolving requirements (Dilhara et al., 2021; Lima et al., 2023; Martins et al., 2024). To automate this process, researchers have developed various approaches based on rules (Moha et al., 2010; Fokaefs et al., 2011; Alharbi and Alshayeb, 2024) or learning mechanisms (Xu et al., 2017; Cui et al., 2023; Liu et al., 2023). However, these technologies face significant hurdles in flexibility and consistency (Azeem et al., 2019). Their heavy reliance on predefined patterns and expert experience often fails to address the complexity of industrial projects, leading to limited adoption by developers (Pantiuchina et al., 2021).

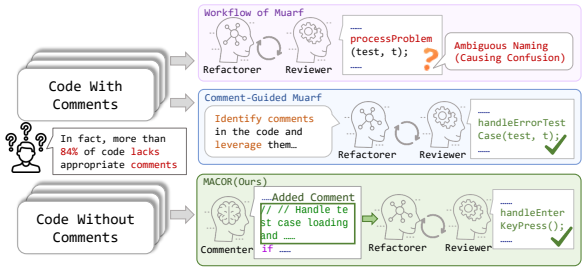

Figure 1: The difference between previous work and our work, **MACOR**.

The emergence of Large Language Models (LLMs) has greatly advanced the development of code refactoring. Current studies have explored using prompt engineering to guide models in executing direct code refactoring. By incorporating human expertise into their working frameworks (Piao et al., 2025), LLMs can replicate natural refactoring patterns that are difficult to capture via rigid rules. Moreover, recent multi-agent frameworks (Shinn et al., 2023; Zhang et al., 2024; Xu, 2025) facilitate collaboration among specialized roles and utilize iterative feedback loops to significantly enhance the reliability and quality of refactored code.

Despite advancements in LLM-driven refactoring, existing approaches often overlook the semantic signals in code comments. Code comments in programs are very important because they record the thoughts and intentions of developers (Perera et al., 2023; Figl et al., 2025). This importance aligns with Fowler's Refactoring Theory, where

* Equal contribution.
† Corresponding author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 32377–32393

the need to add a comment serves as a clear signal for refactoring. By turning descriptions in comments into code structures, comments move the code from low level details to a higher level of abstraction where the code itself explains the business logic. As shown in the top part of Figure 1, using these signals provides necessary guidance to clarify developer goals and complete the refactoring. Without this guidance, Muarf (Xu, 2025) fail to produce optimal results.

While using comments for refactoring is attractive, its practical use is limited by the widespread absence of comments. Writing comments in code is time-consuming and tedious, and developers often neglect this task due to tight development schedules (Chen and Zhou, 2018; Liang and Zhu, 2018). In fact, more than 84 percent of codebases lack appropriate code comments (Huang et al., 2023), which means that merely leveraging existing code comments is insufficient for achieving satisfactory refactoring results in real world projects. Therefore, proactive code generation is necessary for establishing the semantic guidance required for successful refactoring.

To address these challenges, we propose a framework named **MACOR** (a **M**ulti-**A**gent framework for **CO**mment-guided code **R**efactoring). MACOR adopts a proactive generate-then-refactor paradigm, generating precise comments for the original code to support refactoring. Specifically, it identifies key functional points in the original code and generates descriptive comments for these regions, then employs a consistency judgment mechanism to select the optimal comment-augmented version. Subsequently, the code refactoring module leverages these selected signals to retrieve relevant expert examples and guide precise refactoring. Finally, MACOR incorporates an iterative validation mechanism to optimize the output.

We evaluate MACOR across three benchmarks using three LLMs. Experimental results show that MACOR significantly outperforms existing methods. The contributions of this paper are summarized as follows:

- We propose MACOR, a novel multi-agent framework that integrates comment generation into the refactoring process.
- We conduct extensive experiments across multiple datasets to evaluate the performance of MACOR against state-of-the-art baselines.
- Experimental results demonstrate that MACOR significantly outperforms existing methods in both automated metrics and human evaluations.

2 Related Work

2.1 Traditonal Refactoring Approaches

Early automated refactoring mainly relies on rule-based static analysis. Tools such as JDeodorant (Fokaefs et al., 2011) and Decor (Moha et al., 2010) identify code smells by calculating software metrics. However, such methods often result in high failure rates due to rigid rules. To address this issue, search-based software engineering (SBSE) methods (Moghadam and Ó Cinnéide, 2011; Ouni et al., 2012; Cortellessa et al., 2023) model refactoring as a multi-objective optimization problem, searching the solution space for optimal sequences that balance code quality and stability. With the development of software technology, research focus has shifted to data-driven methods. (Aniche et al., 2022; Hamouda et al., 2025) compared the application of different deep learning algorithms in code refactoring tasks. Studies such as (Liu et al., 2021), (Zhang et al., 2022) and (Ma et al., 2023) use deep learning to automatically extract code sequence and structural features for code smell detection. Methods including GEMS (Xu et al., 2017), LiveRef (Fernandes et al., 2023), and REMS (Cui et al., 2023) model code features in various ways to recommend refactoring schemes, reducing the decision-making cost for developers.

2.2 LLM-Based Refactoring Approaches

The rapid advancement of LLMs has enabled them to address complex tasks, offering effective alternatives to traditional refactoring methods. Studies such as InstructCoder (Li et al., 2024) and Octopack (Muennighoff et al., 2024) have enhanced the capacity of LLMs to follow code edit instructions through specialized dataset construction and fine-tuning. Beyond model training, techniques like Chain-of-Thought (CoT) (Wei et al., 2022), few-shot learning (Shirafuji et al., 2023), and EM-Assist (Pomian et al., 2024) utilize prompt engineering to guide LLMs toward superior refactoring outcomes. Other approaches, including PyCraft (Dilhara et al., 2024), Prometheus (Wang et al., 2025a) and iSMELL (Wu et al., 2024), integrate LLM generation with static analysis tools to bolster the reliability of automated refactoring. Recently, research

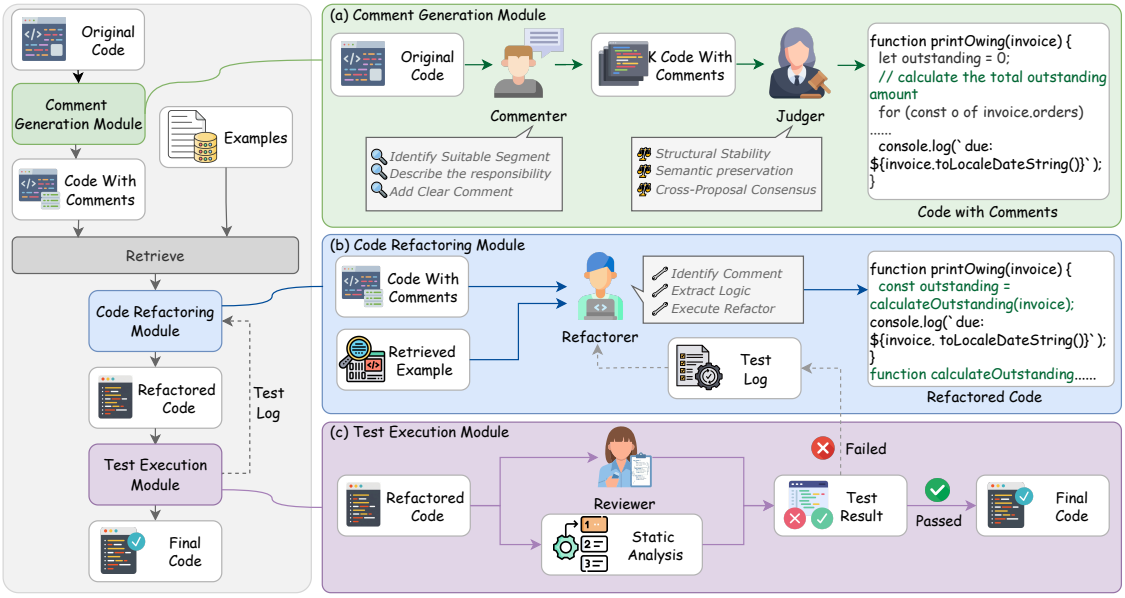

Figure 2: Overview of **MACOR**, which consists of three important modules, (a) the Comment Generation Module, (b) the Code Refactoring Module, and (c) the Test Execution Module.

has pivoted toward multi-agent systems designed to manage intricate contextual information. Autonomous frameworks such as Reflexion (Shinn et al., 2023), AutoCodeRover (Zhang et al., 2024), and Muarf (Xu, 2025) demonstrate sophisticated problem analysis and self-correction, facilitating advanced levels of autonomous software optimization.

3 Approach

3.1 Overview

Figure 2 provides an overview of MACOR, a multi-agent framework based on LLM for comment-guided code refactoring. The system consists of three collaborative modules: the Comment Generation Module, the Code Refactoring Module, and the Test Execution Module. First, the Comment Generation Module produces multiple semantic descriptions using the Commenter. The Judger then selects the most accurate semantic anchor from these proposals based on a consensus mechanism. Next, the Code Refactoring Module utilizes the selected comment-augmented code to retrieve relevant refactoring examples. The Refactorer then generates refactored code by integrating the retrieved examples with the comment guidance. Finally, the Test Execution Module validates the output through similarity comparison and static analysis. If any check fails, the system generates a test log to trigger a self-correction loop.

In the following subsections, we will discuss the Comment Generation Module, the Code Refactoring Module, and the Test Execution Module in detail. For a thorough understanding of MACOR, the detailed process is shown in Algorithm 1 and all the prompt templates are provided in Appendix B.

3.2 Comment Generation Module

The primary objective of the Comment Generation Module is to transform the original code into a comment-augmented version that serves as a precise semantic guide for subsequent refactoring tasks. As illustrated in Figure 2(a), this module focuses on generating high-quality comments to guide the Refactorer in operating effectively.

The process begins with the Commenter, which is tasked with identifying potential functional segments within the messy code and providing concise descriptions of their responsibilities. To ensure the reliability of the generated guidance, the Commenter performs K independent sampling iterations to produce a diverse set of proposals, denoted as $Proposals = \{C_{cmt,1}, \ldots, C_{cmt,K}\}$. Each candidate proposal provides necessary descriptive comments within the code to demonstrate underlying functionality and illuminate regions characterized by semantic uncertainty. These annotations serve as strategic semantic markers that assist the system in localizing and disentangling complex logic.

Algorithm 1 MACOR

Input: Original code C_{orig}, Refactoring example pool E_{pool}, Sampling count K,Max attempts p
Output: Final refactored code C_{final}

// Phase 1: Comment Generation Module
$Proposals \leftarrow \emptyset$
for $i = 1$ **to** K **do**
  $C_{cmt,i} \leftarrow$ Commenter(C_{orig})
  $Proposals \leftarrow Proposals \cup \{C_{cmt,i}\}$
end for
$C_{best\_cmt} \leftarrow$ Judger$(Proposals)$
// Phase 2: Code Refactoring Module
$E_{retrieved} \leftarrow$ Retrieve(C_{best_cmt}, E_{pool})
$Log \leftarrow \emptyset, iter \leftarrow 0$
while $iter < p$ **do**
  $C_{ref} \leftarrow$ Refactorer$(C_{best_cmt}, E_{retrieved}, Log)$
  // Phase 3: Test Execution Module
  $S_{sim} \leftarrow$ SimilarityComparison(C_{ref}, C_{orig})
  $S_{sta} \leftarrow$ StaticAnalysis(C_{ref})
  if $S_{sim} =$ Passed **and** $S_{sta} =$ Passed **then**
    $C_{final} \leftarrow C_{ref}$
    return C_{final}
  else
    $Log \leftarrow$ Feedback(S_{sim}, S_{sta})
    $iter \leftarrow iter + 1$
  end if
end while
$C_{final} \leftarrow C_{ref}$
return C_{final}

Subsequently, the Judger is responsible for selecting the optimal comment-augmented code version. The Judger evaluates the proposals by assessing the alignment between their structural boundaries and semantic intent while identifying the most consistent interpretation across the generated samples. This judgment ensures that the selected code with comments can provide high-quality guidance for subsequent refactoring.

3.3 Code Refactoring Module

The Code Refactoring Module aims to generate high-quality refactored code by leveraging the comments in the code. This module operates by integrating guidance from the comment-augmented code with knowledge derived from external examples. The workflow is divided into the acquisition of retrieved examples and the iterative generation of the refactored code.

First, the module acquires retrieved examples to provide few-shot demonstrations for the Refactorer agent. As illustrated in the framework overview, a retrieval process is executed prior to refactoring, where BM25 is utilized to match the code with comments against a refactoring example pool. This process identifies the most relevant refactoring cases, which serve as technical references for subsequent refactoring operations.

Next, the Refactorer executes the code generation phase. By utilizing both the guidance from the code with comments and the knowledge from the retrieved examples, the Refactorer extracts the target logic into a standalone method with an appropriate name and optimized structure. Furthermore, this generation process is iterative: if the Test Execution Module detects failures, the test log is returned to the Refactorer as feedback. The agent then incorporates this feedback to refine the code in a self-correction loop, ensuring the final result meets all functional and structural requirements within p attempts.

3.4 Test Execution Module

The Test Execution Module serves as a validation gate to ensure that the refactored code maintains functional equivalence with the original version while adhering to structural quality standards. This module employs a dual verification mechanism consisting of a LLM based assessment and Static Analysis to determine whether the refactored code can be the final output.

Reviewer evaluates the consistency between the refactored code and the original code from both semantic and structural perspectives. It analyzes whether the extracted logic faithfully represents the functional intent described in the proactive code comment while ensuring no behavioral changes are introduced during the transformation. This intelligent assessment captures deep semantic alignment that traditional static metrics might miss, providing a more robust verification of functional preservation by reasoning over the logical flow of the program.

In the Static Analysis phase, the system employs the Lizard tool to assess the code health and maintainability. Lizard calculates the Cyclomatic Complexity and Non comment Lines of Code for the refactored method. By enforcing thresholds on these metrics, the system ensures that the extracted method is concise and free of excessive complexity.

The refactored code will only be accepted as the final code if both the Reviewer assessment and static analysis satisfy the predefined criteria. If either check fails, the module generates a comprehensive test log detailing the discrepancies or violations. The test log is then fed back to the Code Refactoring Module, triggering an iterative self correction cycle where the Refactorer attempts to refine the code based on the feedback.

Table 1: Statistics of the three Extract Method datasets.

Statistic	MaRV	EMA-RefA	EMA-RefB
# Samples	88	86	70
Avg. # Lines	46.37	30.42	50.37
Avg. # Tokens	797.35	421.12	897.66

4 Experimental Setup

4.1 Datasets

To comprehensively evaluate our approach, we construct three high-quality Extract Method datasets: **MaRV**, **EMA-RefA**, and **EMA-RefB** based on prior works related to Extract Method refactoring. Each sample contains pre-refactoring and post-refactoring code snippets extracted from real-world Java repositories. The key statistics of datasets are summarized in Table 1.

MaRV: MaRV (Nunes et al., 2025) employs SEART GitHub Search to identify active, popular, and community-driven Java repositories on GitHub and then performs code refactoring mining. Based this dataset, we selected only the samples that received unanimous approval from all reviewers, and filtered out any instances that failed the static analysis checks, resulting in a highly usable Extract Method dataset.

EMA-RefA and **EMA-RefB**: The EMA-Ref datasets are built upon the EM-Assist benchmark, which originally focuses on refactoring location identification for method extraction. While EM-Assist provides metadata and partial context for refactoring, complete post-refactoring code is needed for a comprehensive evaluation of method extraction quality. To this end, we adapted EM-Assist by reconstructing the full pre- and post-refactoring code for each sample from the original repository commit histories. We then performed static analysis and manual verification on the resulting samples to confirm the syntactic completeness and the functional equivalence between pre- and post-refactoring code.

Detailed statistics for the three datasets are provided in Table 1, summarizing the sample counts as well as the average lines and tokens across the benchmarks.

4.2 Evaluation Metrics

We evaluate the refactored code across three dimensions: Consistency, Code Quality, and Human Evaluation.

Consistency Metrics: This dimension measures how closely the refactored code aligns with the original code in terms of semantics and structure.

CodeBLEU (CB): This metric evaluates code similarity by integrating n-gram overlap with syntactic and semantic features. It combines weighted n-grams, Abstract Syntax Tree (AST) matching, and data flow matching.

Tree Edit Distance (TED): This metric quantifies the structural consistency by calculating the minimum number of operations required to transform one AST into another.

Code Quality Metrics: This dimension assesses the improvements in code maintainability and complexity reduction achieved through refactoring. To ensure a rigorous comparison, these metrics are presented as the difference between the model generated code and the human refactored ground truth.

Cyclomatic Complexity (CC): Cyclomatic Complexity is a software metric that quantifies the structural complexity of a program by analyzing its control flow graph (CFG). Specifically, it measures the number of linearly independent paths through the program's control flow, which directly corresponds to the density of decision points in the source code. A lower CC value indicates simpler control logic and enhanced maintainability, whereas higher values suggest greater complexity and potential reliability risks.

Non-comment Lines of Code (NLOC): This metric reflects the size of the source code by counting lines excluding comments and blank lines. It is used to evaluate the efficiency of refactoring in terms of removing redundancy.

Average Function NLOC (AF_NLOC): This metric calculates the average non-comment lines of code per method, reflecting the granularity of method extraction in refactoring. A value closer to that of human-refactored code implies that the model adopts modularization strategies consistent with programmer intuition and reasonable design.

Human Evaluation: This dimension evaluates the practical acceptability and human alignment of the results through a user study involving three computer science graduate students, each having a minimum of 3 years of Java development experience. The evaluators are non-authors of this work and independently rate the samples to avoid bias. We randomly select 50 refactoring samples from three datasets for evaluation. Each sample is evaluated based on two criteria designed to measure functional equivalence and practical usability

respectively.

Expert Alignment (EA): This metric evaluates functional equivalence by comparing model outputs with human standards on a scale from 0 to 4. Scores range from 0 for missing core logic to 4 for a perfect match in all aspects.

Acceptance Score (AS): This indicator measures practical usability based on the reduction of manual effort on a scale from 0 to 4. A score of 0 signifies the output is unusable, while a score of 4 indicates the result is fully usable with negligible manual work.

The detailed scoring criteria are provided in the Appendix A.

4.3 Baselines

To validate the effectiveness of the proposed MACOR for the code refactoring task, we compare MACOR with the following representative baselines:

Direct (DePalma et al., 2024): Directly prompt the LLM to perform code refactoring without any additional techniques.

Few-shot (Shirafuji et al., 2023): Incorporate a few expert examples in the prompt to guide the LLM in refactoring.

COT (Wei et al., 2022): Utilize Chain-of-Thought prompting to encourage the LLM to reason through the refactoring process step-by-step.

Human Experience (Piao et al., 2025): An instruction-driven function that designs procedural strategies for 61 refactoring types based on Martin Fowler's guidelines.

Muarf (Xu, 2025): It automates method-level refactoring through a multi-agent workflow and RAG, integrated with software engineering tools to generate human-like code.

MetaGPT (Hong et al., 2024): A multi-agent framework that encodes Standardized Operating Procedures (SOPs) into workflows to ensure coherent solutions in collaborative software engineering.

4.4 Implementation Details

We benchmark the performance of MACOR on three LLMs: GPT-4o-mini, DeepSeek-v3, and Qwen3-coder-plus, covering both open-source and closed-source models. In the retrieval stage, a semantic similarity-based search mechanism is implemented, with the $top\text{-}k$ value set to 3 to mitigate the potential influence of excessively long contexts. To balance response consistency and diversity, the temperature is set to 0.7 and the maximum response length is limited to 8192 tokens.

5 Experimental Analysis

In this section, we present comprehensive experimental results. Through an in-depth analysis of the results, we aim to address the following Research Questions (**RQs**):

- **RQ1:** How does MACOR perform compared to baseline models in a standard setting?
- **RQ2:** What is the contribution of each key module in MACOR to its overall performance?
- **RQ3:** To what extent does MACOR align with expert developer intuition and provide practical usability in real world scenarios?
- **RQ4:** How does MACOR perform in qualitative evaluations?

5.1 Main Results

To answer **RQ1**, we compare MACOR with several baseline methods on three datasets: emDATA, EMA-RefA, and EMA-RefB. The main results are summarized in Table 2. Additional experiments of adapting comment augmentation only and cross-language generalization can be found in Appendix D.1 and Appendix D.2, respectively.

Enhanced Structural and Semantic Fidelity. MACOR achieves the highest CodeBLEU and TED scores in all tested scenarios, significantly surpassing baseline models. Specifically, on EMA-RefA with DeepSeek-v3, MACOR improves CB by 26.9% and TED by 7.4% relative to the strongest baseline. This superior performance stems from our proactive comment generation, which provides explicit semantic anchors that guide the LLM to reconstruct the code with higher structural and logical fidelity to the ground truth.

Expert-Aligned Quality Optimization. Unlike baselines that often over-simplify or under-refactor code, MACOR yields quality metrics that most closely align with human standards. For instance, on emDATA using GPT-4o-mini, MACOR produces a Cyclomatic Complexity (CC) of 2.02, nearly matching the Ground Truth of 1.99, while baselines deviate significantly with lower complexity. We also computed MACOR's average relative improvement over all baseline methods. Specifically, for each metric, we first compute the performance difference between MACOR and each

Table 2: The Ground Truth rows denote the results of the human-written refactored code, and the **MACOR** rows highlight our proposed method. Average Relative Improvements over Baselines are shown in percentages. Absolute values are provided in the Appendix.

Approaches	MaRV					EMA-RefA					EMA-RefB				
	CB	TED	ΔCC	ΔNLOC	ΔAF_NLOC	CB	TED	ΔCC	ΔNLOC	ΔAF_NLOC	CB	TED	ΔCC	ΔNLOC	ΔAF_NLOC
GPT-4o-mini															
Ground Truth	1.000	1.000	0	0	0	1.000	1.000	0	0	0	1.000	1.000	0	0	0
Direct	1.684	31.614	0.301	4.253	2.758	7.551	2.003	0.794	3.778	4.889	0.424	0.522	2.754	5.614	12.873
Few-shot	1.675	31.904	0.310	4.543	2.830	7.661	2.105	0.800	3.284	4.779	0.415	0.532	2.652	7.414	12.757
COT	1.831	30.036	0.154	2.675	1.961	8.934	2.373	0.554	1.025	3.506	0.435	0.519	2.384	3.757	11.588
Human Exp.	1.696	31.928	0.289	4.567	2.664	7.908	2.084	0.739	3.389	4.532	0.430	0.543	2.673	5.943	12.559
Muarf	1.724	31.036	0.261	3.675	2.597	8.764	2.279	0.610	0.790	3.676	0.435	0.533	2.478	4.336	11.930
MetaGPT	1.699	31.759	0.286	4.398	2.720	7.786	2.025	0.782	2.889	4.654	0.422	0.538	2.732	5.114	12.850
MACOR (ours)	**2.024**	**27.518**	**0.039**	**0.157**	**1.081**	**10.929**	**3.001**	**0.253**	**0.728**	**1.511**	**0.452**	**0.547**	**1.756**	**2.914**	**8.287**
Avg. Rel. Improv.(%)	12.245	7.419	85.384	96.093	58.236	18.519	7.701	64.524	71.178	65.179	5.896	2.981	32.776	45.665	33.310
DeepSeek-v3															
Ground Truth	1.000	1.000	0	0	0	1.000	1.000	0	0	0	1.000	1.000	0	0	0
Direct	0.510	0.612	0.399	7.639	2.917	0.548	0.843	0.896	6.123	5.567	0.417	0.533	2.789	9.957	13.079
Few-shot	0.482	0.602	0.371	9.591	2.962	0.539	0.842	0.906	7.160	5.789	0.418	0.529	2.831	9.771	13.227
COT	0.518	0.640	0.330	5.844	2.438	0.590	0.817	0.690	2.827	4.204	0.422	0.536	2.544	4.943	12.097
Human Exp.	0.520	0.643	0.330	8.350	2.499	0.537	0.852	0.897	7.494	5.423	0.412	0.542	2.801	11.443	12.799
Muarf	0.527	0.645	0.292	7.615	2.233	0.617	0.873	0.639	2.197	3.843	0.430	0.529	2.414	6.164	11.798
MetaGPT	0.493	0.615	0.281	5.000	2.505	0.550	0.840	0.773	3.839	4.812	0.419	0.537	2.557	7.928	12.480
MACOR (ours)	**0.576**	**0.690**	**0.031**	**1.579**	**1.041**	**0.715**	**0.907**	**0.328**	**1.025**	**2.243**	**0.439**	**0.551**	**1.819**	**7.030**	**8.443**
Avg. Rel. Improv.(%)	13.311	10.194	90.714	78.487	59.843	26.886	7.401	59.009	79.251	54.592	4.607	3.119	31.514	15.986	32.886
Qwen3-Coder-Plus															
Ground Truth	1.000	1.000	0	0	0	1.000	1.000	0	0	0	1.000	1.000	0	0	0
Direct	0.565	0.648	0.251	5.434	2.065	0.652	0.874	0.672	3.185	3.911	0.425	0.529	2.507	8.900	11.846
Few-shot	0.539	0.650	0.247	6.735	2.168	0.633	0.880	0.752	5.074	4.741	0.428	0.526	2.392	7.743	11.032
COT	0.525	0.651	0.125	2.374	1.372	0.694	0.862	0.387	0.914	2.555	0.436	0.526	2.014	4.357	9.547
Human Exp.	0.549	0.643	0.170	4.904	1.743	0.644	0.894	0.599	2.667	3.730	0.427	0.530	2.389	8.000	11.344
Muarf	0.602	0.720	0.291	1.145	1.724	0.738	0.905	0.383	0.457	2.617	0.460	0.544	1.910	4.943	9.500
MetaGPT	0.542	0.646	0.246	7.169	2.181	0.646	0.882	0.602	3.111	3.742	0.439	0.543	2.528	7.971	11.862
MACOR (ours)	**0.608**	**0.724**	**0.110**	**0.012**	**0.792**	**0.779**	**0.925**	**0.237**	**0.010**	**1.344**	**0.470**	**0.550**	**1.458**	**1.500**	**7.163**
Avg. Rel. Improv.(%)	9.813	9.752	50.376	99.741	57.771	16.646	4.776	58.115	99.611	62.134	7.839	3.189	36.332	78.527	34.013

baseline, and then report the average of these differences as the relative improvement. In terms of these metrics, our method reduces the NLOC and AF_NLOC gaps by up to 99.74%, demonstrating a unique ability to modularize logic and eliminate redundancy in a manner that closely mimics expert developers.

Cross-Model Robustness. The performance advantage of MACOR is robust across GPT-4o-mini, DeepSeek-v3, and Qwen3-Coder-Plus. This stability across diverse LLM architectures confirms that the benefits of our framework arise from its structured multi-agent collaboration rather than any specific model-specific bias, highlighting its broad applicability in automated software maintenance.

5.2 Abaltion Studies

To answer **RQ2**, we conduct an ablation study across all datasets by evaluating four variants: w/o comment (omitting proactive code generation), w/o judge (removing quality review), w/o tester (excluding functional verification), and w/o rag (disabling retrieval augmentation).

Critical Role of Comment Guidance. The removal of the comment generation module results in the most significant performance degradation. For example, on EMA-RefA, the CodeBLEU drops from 0.715 to 0.556, and TED decreases from 0.907 to 0.846. This sharp decline confirms that comments serve as essential semantic anchors. Without these explicit logical boundaries, the model struggles to maintain structural fidelity, leading to misaligned code transformations that deviate from expert standards.

Contribution of Contextual Knowledge. Disabling the RAG module consistently reduces scores across all datasets. This indicates that retrieving similar refactoring instances provides valuable in-context examples that help the model handle complex method extraction patterns. By providing high-quality reference points, the RAG component enhances the ability of the model to adapt to diverse coding styles.

5.3 Human Evaluation

To answer **RQ3**, we conduct a user study to evaluate the practical quality of refactoring results from a developer perspective. We focus on two critical dimensions: Expert Alignment (EA) to measure the similarity to ground truth and Acceptance Score (AS) to assess usability. The statistical distribution of these scores is illustrated in Figure 3.

Table 3: The Ablation Study of our framework. Results for CC, NLOC, and AF_NLOC are reported as reduction increments (Δ) relative to the source code.

Variants	MaRV					EMA-RefA					EMA-RefB				
	CB	TED	ΔCC	ΔNLOC	ΔAF_NLOC	CB	TED	ΔCC	ΔNLOC	ΔAF_NLOC	CB	TED	ΔCC	ΔNLOC	ΔAF_NLOC
GPT-4o-mini															
w/o comment	0.489	0.598	0.298	4.555	2.656	0.566	0.834	0.837	3.839	4.853	0.418	0.523	2.736	5.714	12.851
w/o judge	0.547	0.638	0.069	1.328	1.181	0.695	0.908	0.273	0.758	1.531	0.437	0.541	1.777	3.114	8.487
w/o tester	0.546	0.647	0.076	1.073	1.248	0.691	0.900	0.294	0.699	1.516	0.450	0.545	2.063	3.143	8.422
w/o rag	0.527	0.631	0.121	0.228	1.370	0.692	0.903	0.258	0.821	1.574	0.451	0.536	1.845	3.043	8.441
MACOR (full model)	**0.550**	**0.654**	**0.039**	**0.157**	**1.081**	**0.704**	**0.909**	**0.253**	**0.728**	**1.511**	**0.452**	**0.547**	**1.756**	**2.914**	**8.287**
DeepSeek-v3															
w/o comment	0.560	0.660	0.276	13.976	2.053	0.556	0.846	0.849	7.444	5.171	0.415	0.524	2.782	11.086	12.959
w/o judge	0.571	0.688	0.051	1.609	1.046	0.703	0.901	0.428	1.045	2.353	0.434	0.535	1.829	7.230	9.143
w/o tester	0.573	0.684	0.116	1.820	1.386	0.701	0.905	0.467	1.173	2.487	0.439	0.536	1.933	7.614	8.584
w/o rag	0.570	0.683	0.181	3.253	1.645	0.684	0.892	0.460	1.164	2.564	0.436	0.542	2.019	7.186	8.897
MACOR (full model)	**0.576**	**0.690**	**0.031**	**1.579**	**1.041**	**0.715**	**0.907**	**0.328**	**1.025**	**2.243**	**0.439**	**0.551**	**1.819**	**7.030**	**8.443**
Qwen3-Coder-Plus															
w/o comment	0.548	0.648	0.197	6.868	1.911	0.653	0.891	0.605	2.926	3.740	0.450	0.529	1.940	4.314	9.276
w/o judge	0.606	0.712	0.150	0.088	0.946	0.763	0.915	0.247	0.040	1.354	0.461	0.538	1.468	1.506	7.263
w/o tester	0.602	0.717	0.184	1.084	1.365	0.765	0.918	0.344	0.272	1.476	0.460	0.543	1.577	1.606	7.216
w/o rag	0.594	0.704	0.140	0.425	1.223	0.769	0.923	0.381	0.383	1.833	0.468	0.545	1.521	2.375	7.513
MACOR (full model)	**0.608**	**0.724**	**0.110**	**0.012**	**0.792**	**0.779**	**0.925**	**0.237**	**0.010**	**1.344**	**0.473**	**0.552**	**1.458**	**1.516**	**7.163**

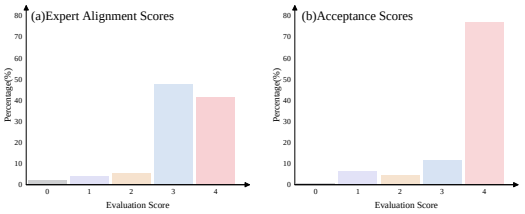

Figure 3: Statistical distribution of human evaluation scores across 50 randomly selected samples.

High Consistency with Expert Logic. As shown in Figure 3(a), MACOR demonstrates a strong capability to align with the design intuition of expert developers. Specifically, most of the refactored samples received scores of 3, indicating that the core method logic and functional implementation are largely consistent with the human written standards. This high alignment confirms that the proactive generation of a code comment effectively serves as a semantic anchor, helping the model identify logical boundaries that would otherwise be ambiguous in original code.

Superior Practical Usability. The Acceptance Score in Figure 3(b) highlights the significant reduction in manual development effort provided by our framework. Notably, nearly 80 percent of the samples achieved a full score of 4, meaning the refactoring results are directly usable with almost no manual modifications. Only a negligible fraction of samples were rated as completely unusable, which typically occurred in extremely complex cases with deep nested logic where even an automated code comment could not fully resolve structural ambiguity. These results suggest that MACOR is highly effective for industrial applications where developers face tight schedules and lack the time to write a code comment manually.

5.4 Case Study

To answer **RQ4**, we present a representative example from EMA-RefA to illustrate how MACOR operates in a realistic refactoring scenario. As shown in Figure 4, the original *keyTyped* method embeds the logic for handling the Enter key together with other key-event processing logic in a single method. This coupled implementation makes the boundary of the target functionality less explicit, which in turn increases the difficulty of directly extracting it into a separate method.

Comment-Enhanced Candidate Selection. MACOR first generates multiple candidate refactoring versions for the original code. Each candidate is paired with a concise code comment that explicitly describes the intended behavior of the logic to be extracted. These comments make the purpose and scope of each candidate easier to understand and compare. The judger then evaluates the commented candidates and selects the version that best matches the target refactoring intent, providing a clearer starting point for the subsequent transformation.

Comment-Guided Refactoring Outcome. In the refactoring stage, MACOR retrieves a similar example that demonstrates an appropriate method-delegation pattern. Guided by this example, the refactorer separates the Enter-key handling logic from *keyTyped* and reorganizes it into *handleEn-*

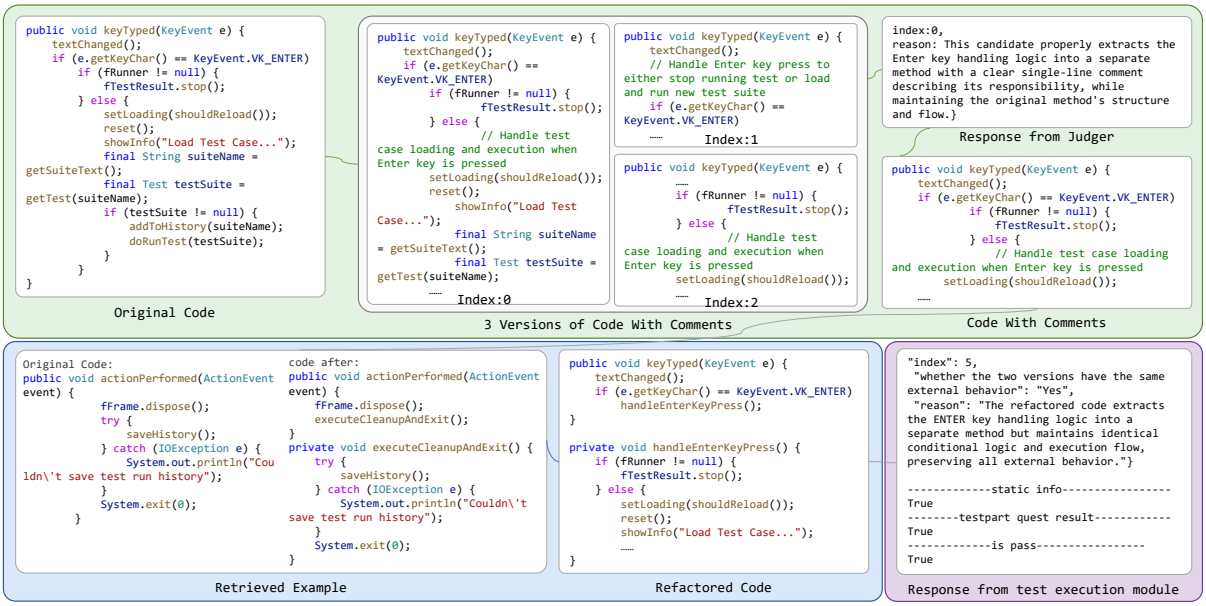

Figure 4: Case study from EMA-RefA.

terKeyPress. As a result, the original coupled implementation is transformed into a clearer delegated structure, in which *keyTyped* retains the overall event-processing flow while the extracted method is responsible for the specific Enter-key behavior. Finally, the test execution module verifies that the refactored code preserves both the expected structure and the original behavior.

6 Conclusion

In this paper, we introduced MACOR, a comment-guided multi-agent framework designed for automated code refactoring. By generating structural comments as proactive logical guides, MACOR not only achieves enhanced structural and semantic fidelity but also ensures expert-aligned quality optimization. Furthermore, we enhanced the framework through a judger for comment validation and a test execution module to provide iterative feedback based on static analysis and structural similarity. Experimental results across three datasets and various LLMs validate the effectiveness of MACOR, demonstrating its superior performance in autonomous software maintenance.

Limitations

We identify three key limitations in MACOR as follows:

Lack of Processing for Obsolete Legacy Comments. While MACOR proactively generates new structural comments to guide the refactoring process, it currently lacks a dedicated mechanism to identify conflicts with existing obsolete or incorrect comments in the source code. The absence of a pre-processing step to filter such legacy noise may introduce semantic interference during intent extraction, potentially hindering the framework from achieving optimal enhanced structural and semantic fidelity.

Unoptimized Retrieval Contexts. The retrieval augmented component of our framework utilizes samples directly extracted from datasets without employing meticulous curation or fine-grained heuristic selection. This reliance on unoptimized contexts may limit the capacity of the framework to provide the most relevant architectural guidance for complex refactoring scenarios, which remains a key objective for further Expert-Aligned Quality Optimization.

Restricted Structural Scope. Current evaluations of MACOR are primarily restricted to method-level refactoring within individual source files. The framework does not yet incorporate mechanisms for repository-level transformations that require global dependency tracing and cross-file structural adjustments.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments. This work was supported by the National Natural Science Foundation of China (62176053).

References

Maha Alharbi and Mohammad Alshayeb. 2024. A comparative study of automated refactoring tools. *IEEE Access*, 12:18764–18781.

Maurício Aniche, Erick Maziero, Rafael Durelli, and Vinicius H. S. Durelli. 2022. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering*, 48(4):1432–1450.

Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, 108:115–138.

Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 826–831.

Vittorio Cortellessa, Daniele Di Pompeo, Vincenzo Stoico, and Michele Tucci. 2023. Many-objective optimization of non-functional attributes based on refactoring of software models. *Information and Software Technology*, 157:107159.

Di Cui, Qiangqiang Wang, Siqi Wang, Jianlei Chi, Jianan Li, Lu Wang, and Qingshan Li. 2023. Rems: Recommending extract method refactoring opportunities via multi-view representation of code property graph. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*, pages 191–202.

Kayla DePalma, Izabel Miminoshvili, Chiara Henselder, Kate Moss, and Eman Abdullah AlOmar. 2024. Exploring chatgpt's code refactoring capabilities: An empirical study. *Expert Syst. Appl.*, 249(PB).

Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, and Danny Dig. 2024. Unprecedented code change automation: The fusion of llms and transformation by example. *Proc. ACM Softw. Eng.*, 1(FSE).

Malinda Dilhara, Ameya Ketkar, and Danny Dig. 2021. Understanding software-2.0: A study of machine learning library usage and evolution. *ACM Trans. Softw. Eng. Methodol.*, 30(4).

Sara Fernandes, Ademar Aguiar, and André Restivo. 2023. Liveref: a tool for live refactoring java code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA. Association for Computing Machinery.

Kathrin Figl, Maria Kirchner, Sebastian Baltes, and Michael Felderer. 2025. The influence of code comments on the perceived helpfulness of stack overflow posts. *Empirical Softw. Engg.*, 30(6).

Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2011. Jdeodorant: identification and application of extract class refactorings. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1037–1039.

Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Object Technology Series. Addison-Wesley.

Esraa Hamouda, Abeer El-Korany, and Soha Makady. 2025. Smell-ml: A machine learning framework for detecting rarely studied code smells. *IEEE Access*, 13:12966–12980.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

Yuan Huang, Hanyang Guo, Xi Ding, Junhuai Shu, Xiangping Chen, Xiapu Luo, Zibin Zheng, and Xiaocong Zhou. 2023. A comparative study on method comment and inline comment. *ACM Trans. Softw. Eng. Methodol.*, 32(5).

Kaixin Li, Qisheng Hu, James Xu Zhao, Hui Chen, Yuxi Xie, Tiedong Liu, Michael Shieh, and Junxian He. 2024. InstructCoder: Instruction tuning large language models for code editing. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pages 473–493, Bangkok, Thailand. Association for Computational Linguistics.

Yuding Liang and Kenny Zhu. 2018. Automatic generation of text descriptive comments for code blocks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1).

Danilo Leandro Lima, Ronnie De Souza Santos, Guilherme Pires Garcia, Sildemir S. Da Silva, Cesar França, and Luiz Fernando Capretz. 2023. Software testing and code refactoring: A survey with practitioners. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 500–507.

Bo Liu, Hui Liu, Guangjie Li, Nan Niu, Zimao Xu, Yifan Wang, Yunni Xia, Yuxia Zhang, and Yanjie Jiang. 2023. Deep learning based feature envy detection boosted by real-world examples. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 908–920, New York, NY, USA. Association for Computing Machinery.

Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. 2021. Deep learning based code smell detection. *IEEE Transactions on Software Engineering*, 47(9):1811–1837.

Wenhao Ma, Yaoxiang Yu, Xiaoming Ruan, and Bo Cai. 2023. Pre-trained model based feature envy detection. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 430–440.

Luana Martins, Valeria Pontillo, Heitor Costa, Filomena Ferrucci, Fabio Palomba, and Ivan Machado. 2024. Test code refactoring unveiled: where and how does it affect test code quality and effectiveness? *Empirical Softw. Engg.*, 30(1).

Iman Hemati Moghadam and Mel Ó Cinnéide. 2011. Code-imp: a tool for automated search-based refactoring. In *Proceedings of the 4th Workshop on Refactoring Tools*, WRT '11, page 41–44, New York, NY, USA. Association for Computing Machinery.

Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2010. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36.

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations*.

Henrique Nunes, Tushar Sharma, and Eduardo Figueiredo. 2025. Marv: A manually validated refactoring dataset. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, pages 141–145.

Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mohamed Salah Hamdi. 2012. Search-based refactoring: Towards semantics preservation. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 347–356.

Jevgenija Pantiuchina, Bin Lin, Fiorella Zampetti, Massimiliano Di Penta, Michele Lanza, and Gabriele Bavota. 2021. Why do developers reject refactorings in open-source projects? *ACM Trans. Softw. Eng. Methodol.*, 31(2).

D. T. W. S. Perera, H. T. M. Premathilake, K. P. H. Thathsarani, R. H. T. Nethmini, D. I. De Silva, and H. M. P. P. K. H. Samarasekara. 2023. Analyzing the impact of code commenting on software quality. In *2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–6.

Yonnel Chen Kuang Piao, Jean Carlors Paul, Leuson Da Silva, Arghavan Moradi Dakhel, Mohammad Hamdaqa, and Foutse Khomh. 2025. Refactoring with llms: Bridging human expertise and machine understanding. *Preprint*, arXiv:2510.03914.

Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Andrey Sokolov, Timofey Bryksin, and Danny Dig. 2024. Em-assist: Safe automated extractmethod refactoring with llms. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 582–586, New York, NY, USA. Association for Computing Machinery.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

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

Sewen Thy, Andreea Costea, Kiran Gopinathan, and Ilya Sergey. 2023. Adventure of a lifetime: Extract method refactoring for rust. *Proc. ACM Program. Lang.*, 7(OOPSLA2).

Changjie Wang, Mariano Scazzariello, Anoud Alshnakat, Roberto Guanciale, Dejan Kostić, and Marco Chiesa. 2025a. Dissect-and-restore: Ai-based code verification with transient refactoring. *Preprint*, arXiv:2510.25406.

Siqi Wang, Xing Hu, Xin Xia, and Xinyu Wang. 2025b. Actref: Enhancing the understanding of python code refactoring with action-based analysis. *arXiv preprint arXiv:2505.06553*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

Di Wu, Fangwen Mu, Lin Shi, Zhaoqiang Guo, Kui Liu, Weiguang Zhuang, Yuqi Zhong, and Li Zhang. 2024. ismell: Assembling llms with expert toolsets for code smell detection and refactoring. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1345–1357, New York, NY, USA. Association for Computing Machinery.

Sihan Xu, Aishwarya Sivaraman, Siau-Cheng Khoo, and Jing Xu. 2017. Gems: An extract method refactoring recommender. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 24–34.

Yisen Xu. 2025. Muarf: Leveraging multi-agent workflows for automated code refactoring. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 226–227.

Yang Zhang, Chuyan Ge, Shuai Hong, Ruili Tian, Chunhao Dong, and Jingjing Liu. 2022. Delesmell: Code smell detection based on deep learning and latent semantic analysis. *Know.-Based Syst.*, 255(C).

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1592–1604, New York, NY, USA. Association for Computing Machinery.

A Scoring Criteria for Human Evaluation

We design a set of scoring criteria to evaluate the functional equivalence and practical usability of the refactoring results generated by the model. The criteria details are shown in Figure 5.

The final score for each sample is averaged across the three evaluators. Discrepancies in scoring are discussed to reach a consensus, ensuring the reliability and consistency of the evaluation.

B Prompt Templates

In this section, we present the prompt templates employed for MACOR. To facilitate a clearer presentation, we organize the prompts for the four agents into a structured format as shown in Figure 6.

C Raw Software Quality Metrics of Refactoring Results

To provide a comprehensive view of the software quality after refactoring, Table 4 presents the absolute values of Cyclomatic Complexity (CC), Number of Lines of Code (NLOC), and Average Functional NLOC (AF_NLOC) for all evaluated models and benchmarks.

While the main results in Table 2 focus on the reduction increments (Δ) to highlight the relative improvements of our proposed method, the absolute metrics provided here serve as a complete reference for the structural properties of the generated code.

D Additional Experimental Results

D.1 Comment Augmentation

To ensure the rigor of our framework, we conducted an extended study to determine whether the performance gains are uniquely tied to the MACOR architecture or whether baseline models can achieve comparable results through simple comment augmentation. This investigation helps clarify the role of explicit semantic guidance in the automated refactoring process.

We augmented several representative baselines with the same LLM-generated comments used in our framework and evaluated them across all three datasets on GPT-4o-mini. During the above procedure, comments are generated using the same Commenter module in MACOR. Specifically, given the original code without comments, we first apply the Commenter to generate explanatory comments. The resulting commented code is then used as the input for the corresponding baseline method, and all subsequent operations are performed on this commented code. For example, Direct+Comm. denotes directly refactoring the code after augmenting it with comments, using the same refactoring prompt as in the original Direct baseline. The results are summarized in Table 6.

Our observations indicate that while the inclusion of comments consistently improves the consistency (CodeBLEU and TED) and reduces the complexity (CC and NLOC) for most baselines, MACOR continues to outperform these augmented versions in nearly all categories. This suggests that while semantic comments provide a foundational scaffold for the LLM, the superior efficacy of our approach is derived from the coordinated multi-agent workflow and the iterative feedback loop, which leverage these comments more effectively than traditional prompting strategies.

D.2 Generalization Across Programming Languages

While our primary evaluations are conducted on Java, the architecture of MACOR is designed to be language-agnostic, as it relies on natural-language semantic anchors rather than language-specific static analysis rules.

To verify the cross-language generalization of our framework, we conducted supplemental experiments on Python and Rust. For Python, we utilized the ActRef benchmark (Wang et al., 2025b), a recent dataset focused on action-based analysis of Python refactoring. For Rust, we collected 25 refactoring instances from the dataset provided by REM(Thy et al., 2023) and followed the same experimental protocol as our main study.

The results summarized in Table 7, demonstrate that MACOR consistently outperforms representative baselines across both languages. Notably,

Criterion 1: Functional Equivalence to Human Refactoring
This criterion measures how closely the model's refactored code aligns with the human-refactored version, with higher scores indicating better functional consistency.

0 point: The model's refactored code severely lacks core method logic present in the human version, posing a significant threat to the primary functionality of the code.

1 point: Major functional components are missing compared to the human refactoring, leading to substantial deviations in behavior.

2 point: Moderate mismatches exist, such as incomplete implementation of core functions or notable structural differences.

3 point: Minor differences are present, such as variable renaming, or additions/omissions of non-essential code segments, while primary functionality is fully preserved.

4 point: The model's refactored code closely matches the human-refactored code in both logic and structure.

Criterion 2: Practical Usability
This criterion evaluates whether the refactoring result can be directly used in development to reduce manual effort.

0 point: Completely unusable—the evaluator would prefer to manually refactor the code from scratch.

1 point: Partially usable—provides only limited assistance in reducing refactoring effort.

2 point: Mostly usable—can help reduce a noticeable portion of human refactoring work.

3 point: Usable—lowers human effort to a low degree, requiring only minor adjustments.

4 point: Fully usable—the refactored sample can be adopted directly with negligible additional human effort.

Figure 5: The criteria details for human evaluation.

in Python and Rust settings, MACOR achieves superior scores in consistency metrics (CodeBLEU, TED) and shows a significant advantage in reducing code complexity (ΔCC and ΔNLOC).These findings suggest that the effectiveness of comment-guided refactoring is not limited to a single syntax or tooling ecosystem.

By leveraging LLM-driven reasoning and example retrieval, MACOR exhibits promising applicability to diverse programming environments, including those where large-scale refactoring corpora or advanced static analysis tools may be less accessible.

D.3 Analysis of Multi-stage Workflow vs. Single-step Baseline

To investigate whether the performance gains of MACOR could be achieved through a more efficient single-step formulation, such as using Chain-of-Thought (CoT) prompting to generate comments and refactored code in a single model call, we conducted a comparative study against a Single-step Baseline, where a single model call is prompted to generate semantic comments, perform internal self-evaluation, and produce the final refactored code.

The results conducted on GPT-4o-mini are summarized in Table 8. Across all three datasets, the single-step approach consistently underperforms the multi-stage pipeline of MACOR.

Based on our analysis, the advantages of the decomposed workflow stem from two primary factors:

Complexity Control: In a single-step formulation, the model must handle substantially longer outputs within one generation context. As the output length increases, the model's instruction-following capability and coherence tend to degrade, leading to logical inconsistencies. By decoupling these tasks, MACOR constrains the complexity of each individual step, ensuring more stable generation.

Mitigation of Self-Correction Bias: When evaluation is merged into the same generation step as content creation, models often exhibit an overly optimistic bias, allowing underspecified comments to pass implicit validation. MACOR enforces a separate, explicit evaluation stage, which provides a more rigorous filter for semantic guidance before it influences the refactoring process.

E Case Study of Refactoring Failure

To illustrate the risks discussed in the previous section, we present a specific failure case (Figure 7)

Prompt for Commenter

```
Identify the most suitable code segment for
method extraction in the following code,
and add a SINGLE LINE comment that clearly
describes the responsibility of that segment,
return the whole code after adding comment.

## code:
{code_before}

## Important:
You should give the whole code after adding
comment. Do not add extra explanation or words.
Your answer must inside "‘ block.
```

Prompt for Refactorer

```
Perform method extraction refactoring on
the following code. The SINGLE LINE Comment
describing responsibility of a complex
code segment may inference the need of the
extraction refactoring.

## Refactoring Example:
{similar_refactoring_example}

## Code need to be refactored:
{code_with_comment}

## Important:
You should give the refactored code only. Do
not add extra explanation or words.
```

Prompt for Judger

```
Given several candidates suggestion for
processing extract-method refactoring on code
snippets...

## Candidates:
{candidate_str}

Return the index of the candidate.

## Important:
Return one index in range of 0 to
{candidate_length-1}, and a simple reason, for
example:
{index: <index>, reason: <reason>}
```

Prompt for Reviewer

```
Given the code before and after refactoring,
please analyze whether the two versions have
the same external behavior. Please give a
brief reason.

## code before refactoring:
{code_before}

## code after refactoring:
{code_after_refact}

## Important:
Your answer should follow the format:
{
  "behavior": "Yes"/"No",
  "reason": "<brief reason>"
}
```

Figure 6: Detailed prompt templates for the four agents (Commenter, Judger, Refactorer, and Reviewer) in the MACOR framework.

where a hallucinated or misinterpreted semantic anchor led to incorrect refactoring.

In this instance, the Commenter generated a note stating that it is "not possible to retrieve info from a Uri alone." Although this was intended as a descriptive observation, the Refactor agent misinterpreted this as a bug that required a functional "fix." Consequently, it introduced non-existent external dependencies (e.g., FileCache and Log) and altered the return logic, which deviated from the original program's behavior.

This case confirms that while the framework is generally robust, semantic anchors that introduce external assumptions can lead to logical drifts, which our Test Execution Module is designed to capture.

Table 4: Absolute values of software quality metrics (CC, NLOC, and AF_NLOC) across three benchmarks. The Ground Truth rows denote human-written code, and **MACOR** rows highlight our method.

Approaches	MaRV			EMA-RefA			EMA-RefB		
	CC	NLOC	AF_NLOC	CC	NLOC	AF_NLOC	CC	NLOC	AF_NLOC
GPT-4o-mini									
Ground Truth	1.985	27.361	8.666	2.562	25.926	12.440	4.757	41.143	20.503
Direct	1.684	31.614	5.908	1.768	29.704	7.551	2.003	46.757	7.630
Few-shot	1.675	31.904	5.836	1.762	29.210	7.661	2.105	48.557	7.746
COT	1.831	30.036	6.705	2.008	26.951	8.934	2.384	44.900	8.915
Human Exp.	1.696	31.928	6.002	1.823	29.315	7.908	2.084	47.086	7.944
Muarf	1.724	31.036	6.069	1.952	26.716	8.764	2.279	45.479	8.573
MetaGPT	1.699	31.759	5.946	1.780	28.815	7.786	2.025	46.257	7.653
MACOR (ours)	**2.024**	**27.518**	**7.585**	**2.309**	**25.198**	**10.929**	**3.001**	**44.057**	**12.216**
DeepSeek-v3									
Ground Truth	1.985	27.361	8.666	2.562	25.926	12.440	4.757	41.143	20.503
Direct	1.586	35.000	5.749	1.666	32.049	6.873	1.968	51.100	7.424
Few-shot	1.614	36.952	5.704	1.656	33.086	6.651	1.926	50.914	7.276
COT	1.655	33.205	6.228	1.872	28.753	8.236	2.213	46.086	8.406
Human Exp.	1.655	35.711	6.167	1.665	33.420	7.017	1.956	52.586	7.704
Muarf	1.693	34.976	6.433	1.923	28.123	8.597	2.343	47.307	8.705
MetaGPT	1.704	32.361	6.161	1.789	29.765	7.628	2.200	49.071	8.023
MACOR (ours)	**1.954**	**28.940**	**7.625**	**2.234**	**26.951**	**10.197**	**2.938**	**48.173**	**12.060**
Qwen3-Coder-Plus									
Ground Truth	1.985	27.361	8.666	2.562	25.926	12.440	4.757	41.143	20.503
Direct	1.734	32.795	6.601	1.890	29.111	8.529	2.250	50.043	8.657
Few-shot	1.738	34.096	6.498	1.810	31.000	7.699	2.365	48.886	9.471
COT	1.860	29.735	7.294	2.175	26.840	9.885	2.743	45.500	10.956
Human Exp.	1.815	32.265	6.923	1.963	28.593	8.710	2.368	49.143	9.159
Muarf	1.694	28.506	6.942	2.179	26.383	9.823	2.847	46.086	11.003
MetaGPT	1.739	34.530	6.485	1.960	29.037	8.698	2.229	49.114	8.641
MACOR (ours)	**1.875**	**27.349**	**7.874**	**2.325**	**25.916**	**11.096**	**3.299**	**42.643**	**13.340**

Table 5: Absolute values of software quality metrics for the Ablation Study across three benchmarks. The **MACOR** rows represent our full model implementation.

Variants	MaRV			EMA-RefA			EMA-RefB		
	CC	NLOC	AF_NLOC	CC	NLOC	AF_NLOC	CC	NLOC	AF_NLOC
GPT-4o-mini									
w/o comment	1.687	31.916	6.010	1.725	29.765	7.587	2.021	46.857	7.652
w/o judge	2.054	28.689	7.485	2.289	25.168	10.909	2.980	44.257	12.016
w/o tester	2.061	28.434	7.418	2.268	25.227	10.924	2.694	44.286	12.081
w/o rag	2.106	27.133	7.296	2.304	25.105	10.866	2.912	44.186	12.062
MACOR (full model)	**2.024**	**27.518**	**7.585**	**2.309**	**25.198**	**10.929**	**3.001**	**44.057**	**12.216**
DeepSeek-v3									
w/o comment	1.709	41.337	6.613	1.713	33.370	7.269	1.975	52.229	7.544
w/o judge	1.934	28.970	7.620	2.134	26.971	10.087	2.928	48.373	11.360
w/o tester	1.869	29.181	7.280	2.095	27.099	9.953	2.824	48.757	11.919
w/o rag	1.804	30.614	7.021	2.102	27.090	9.876	2.738	48.329	11.606
MACOR (full model)	**1.954**	**28.940**	**7.625**	**2.234**	**26.951**	**10.197**	**2.938**	**48.173**	**12.060**
Qwen3-Coder-Plus									
w/o comment	1.790	34.230	6.760	1.960	28.850	8.700	2.820	45.460	11.230
w/o judge	1.870	27.540	7.720	2.450	26.010	11.690	3.260	42.790	13.210
w/o tester	1.800	26.280	7.300	2.440	26.200	11.460	3.380	42.030	14.490
w/o rag	1.850	27.390	7.440	2.480	25.540	11.510	3.240	38.770	12.990
MACOR (full model)	**1.880**	**27.350**	**7.870**	**2.330**	**25.940**	**11.100**	**3.300**	**42.640**	**13.340**

Table 6: Generality analysis of comment augmentation, comparing baselines with and without comments against MACOR on GPT-4o-mini. Results cover consistency metrics and quality metric variations (Δ).

Approach	MaRV					EMA-RefA					EMA-RefB				
	CB ↑	TED ↑	ΔCC ↓	ΔNL ↓	ΔAF ↓	CB ↑	TED ↑	ΔCC ↓	ΔNL ↓	ΔAF ↓	CB ↑	TED ↑	ΔCC ↓	ΔNL ↓	ΔAF ↓
Direct	0.487	0.602	0.301	4.253	2.758	0.564	0.827	0.794	3.778	4.889	0.424	0.522	2.754	5.614	12.873
Direct+Comm.	0.502	0.625	0.125	1.457	0.979	0.608	0.865	0.313	0.963	2.886	0.413	0.509	1.892	3.676	9.025
Few-shot	0.473	0.606	0.310	4.543	2.830	0.563	0.849	0.800	3.284	4.779	0.415	0.532	2.652	7.414	12.757
Few-shot+Comm.	0.485	0.623	0.050	0.301	1.602	0.579	0.861	0.522	0.136	3.568	0.402	0.535	2.182	5.497	10.604
COT	0.512	0.614	0.154	2.675	1.961	0.635	0.825	0.554	1.025	3.506	0.435	0.519	2.384	3.757	11.588
COT+Comm.	0.518	0.642	0.224	0.891	0.420	0.617	0.869	0.336	1.358	1.924	0.411	0.522	1.898	2.964	6.698
Human Exp.	0.491	0.604	0.289	4.567	2.664	0.600	0.863	0.739	3.389	4.532	0.430	0.543	2.673	5.943	12.851
Human Exp.+Comm.	0.512	0.641	0.049	0.240	1.106	0.619	0.887	0.410	1.173	2.795	0.403	0.544	1.925	5.310	9.163
Muarf	0.499	0.622	0.261	3.675	2.597	0.630	0.861	0.610	0.790	3.676	0.435	0.533	2.478	4.336	11.930
Muarf+Comm.	0.513	0.646	0.205	0.503	1.970	0.637	0.872	0.396	0.161	2.558	0.452	0.530	1.873	6.090	9.082
MACOR (ours)	**0.550**	**0.654**	**0.039**	**0.157**	**1.081**	**0.704**	**0.909**	**0.253**	**0.728**	**1.511**	**0.452**	**0.547**	**1.756**	**2.914**	**8.287**

Table 7: The performance comparison on Python (ActRef) and Rust datasets on GPT-4o-mini.

Approaches	Python (ActRef)					Rust				
	CB ↑	TED ↑	ΔCC ↓	ΔNL ↓	ΔAF ↓	CB ↑	TED ↑	ΔCC ↓	ΔNL ↓	ΔAF ↓
Direct	0.537	0.675	0.236	21.634	2.007	0.738	0.922	0.444	21.840	5.756
Few-shot	0.502	0.627	0.336	21.467	3.078	0.728	0.915	0.281	11.080	5.523
COT	0.524	0.649	0.304	27.155	2.082	0.709	0.879	0.656	18.480	5.529
Human Exp.	0.558	0.686	0.403	21.978	2.228	0.756	0.931	0.299	28.200	7.031
Muarf	0.511	0.648	0.138	24.124	1.779	0.743	0.926	0.348	22.360	7.164
MACOR (ours)	**0.594**	**0.709**	**0.207**	**15.780**	**1.777**	**0.782**	**0.934**	**0.255**	**4.260**	**5.440**

Table 8: Performance comparison between the proposed multi-stage MACOR and a single-step baseline on GPT-4o-mini. Results cover consistency metrics and quality metric variations (Δ).

Approach	MaRV					EMA-RefA					EMA-RefB				
	CB ↑	TED ↑	ΔCC ↓	ΔNL ↓	ΔAF ↓	CB ↑	TED ↑	ΔCC ↓	ΔNL ↓	ΔAF ↓	CB ↑	TED ↑	ΔCC ↓	ΔNL ↓	ΔAF ↓
Single-Step	0.477	0.595	0.396	2.108	1.220	0.634	0.861	0.316	0.975	1.991	0.448	0.524	1.776	6.303	8.418
MACOR (ours)	**0.550**	**0.654**	**0.039**	**0.157**	**1.081**	**0.704**	**0.909**	**0.253**	**0.728**	**1.511**	**0.452**	**0.547**	**1.756**	**2.914**	**8.287**

Original Code

```
public class TempClass {
  private AtomicBoolean
       reportRunningFlag = new
       AtomicBoolean(false);
  /** Deletes the file represented by
       given Uri */
  public boolean delete(final Uri uri)
       {
    if (uri == null) {
      return false;
    }
    try {
      deleteFile(uri);
      return true;
    } catch (Exception e) {
      return false;
    } finally {
      reportRunningFlag.set(false);
    }
  }
  public FileInformation
       getFileInfo(final Uri uri) {
    if (uri == null) {
      return null;
    }
    getFileInformation(uri);
    return null;
  }
  ......
}
```

Code with Comments

```
public class TempClass {
  private AtomicBoolean
       reportRunningFlag = new
       AtomicBoolean(false);
  /** Deletes the file represented by
       given Uri */
  public boolean delete(final Uri uri)
       {
    if (uri == null) {
      return false;
    }
    try {
      deleteFile(uri);
      return true;
    } catch (Exception e) {
      return false;
    } finally {
      reportRunningFlag.set(false);
    }
  }
  // Note: it is not possible to
       retrieve this info from an Uri
       alone.
  public FileInformation
       getFileInfo(final Uri uri) {
    if (uri == null) {
      return null;
    }
    getFileInformation(uri);
    return null;
  }
  ......
}
```

Incorrectly Refactored Code

```
  public class TempClass {
  private AtomicBoolean
       reportRunningFlag = new
       AtomicBoolean(false);
  public FileInformation
       getFileInfo(final Uri uri) {
    if (isEmpty(uri)) {
      return null;
    }
    Log.w("TempClass", "Cannot retrieve
         info from Uri alone as per
         documentation.");
    FileInformation cachedInfo =
         FileCache.get(uri.toString());
    if (cachedInfo != null) {
      return cachedInfo;
    }
    return new
         FileInformation("Unknown", 0);
  }
  ......
}
```

Figure 7: A failure case study. In this case the LLM misinterpreted a descriptive comment as an actionable problem, leading to the introduction of non-existent interfaces and logic drifts.