Leveraging Fine-Tuned Large Language Models for Personalized Feedback in Undergraduate Object-Oriented Programming Education

Zuchen Li[1] Ji Wu[1,2] Qitong Liu[3] Qing Sun*[1,2] Jing Li[2,4] Wenge Rong[1,2]
[1] *School of Computer Science and Engineering, Beihang University, Beijing, China*
[2] *Engineering Research Center of Integration and Application of Digital Learning Technology, Ministry of Education, China*
[3] *School of Cyber Science and Technology, Beihang University, Beijing, China*
[4] *The Open University of China, Beijing, China*
Email: {lizuchen, wuji, liuqt, sunqing, w.rong}@buaa.edu.cn, lijing2015@ouchn.edu.cn

***Abstract*—In undergraduate software engineering education, architectural design courses are pivotal for fostering higher-order engineering skills through complex, iterative projects. However, providing effective feedback in these courses remains challenges: traditional static analysis verifies functional correctness but cannot evaluate architectural quality, while manual review is unscalable for large student cohorts. To address these challenges, this paper proposes a generalizable LLM-based feedback framework, named Helper, which integrates three core modules: course-adaptable data preparation; lightweight fine-tuning for general models; and context-aware recommendation generation. This paper also presents a dedicated fine-tuning dataset tailored to the Object-Oriented Design (OOD) course. The Helper's specific instantiation of the OOD course is named OO-Helper. Experimental validation demonstrates that OO-Helper outperforms general foundation models in modification accuracy and design compliance. Qualitative feedback from students confirms that it can act as an effective pedagogical scaffold. The modular framework enables seamless cross-disciplinary migration with minor adjustments, offering broad practical value for CS education.**

***Index Terms*—Automated Feedback, Education Technology, LoRA Fine-tuning, Object-Oriented Design**

I. Introduction

In undergraduate software engineering education, architectural design courses are pivotal for fostering higher-order engineering skills through complex, iterative projects [1]. However, providing effective feedback in this context remains a persistent challenge [2]. Traditional manual review is unscalable for large cohorts, while static analysis tools primarily focus on functional correctness and fail to evaluate architectural quality or provide constructive design guidance [3]. Large Language Models (LLMs) have emerged as promising intelligent tutors [4]. Despite their general coding capabilities, off-the-shelf LLMs often lack domain-specific knowledge—such as specific course requirements and iterative development contexts—leading to generic advice or hallucinations [5]. While techniques like Retrieval-Augmented Generation and Supervised Fine-Tuning can incorporate external knowledge [6], applying them to architectural design education is non-trivial. First, architectural design is open-ended with no single "gold standard," making it difficult to construct a consistent knowledge base. Second, raw student code contains excessive implementation details that obscure the high-level structural evolution required for effective model learning.

* Corresponding author: Qing Sun (email: sunqing@buaa.edu.cn).

To bridge the gap between abstract design principles and concrete implementation, we propose Helper, a generalizable LLM-based feedback framework. The framework integrates three core modules: (1) course-adaptable data preparation, employing static analysis-driven UML extraction for noise filtering and teaching rule-guided annotation; (2) Low-Rank Adaptation (LoRA) lightweight fine-tuning for general models such as Qwen3-14B; and (3) context-aware recommendation generation that guides LLMs to prioritize structural evolution over syntax. Additionally, we build a specialized Object-Oriented Design dataset, and we evaluate our approach in a real-world Object-Oriented Design course (OOD). Our main contributions are:

- **A Generalizable Helper Framework for Automated Architectural Feedback:** We propose the Helper framework, which is inherently generalizable to diverse software engineering courses and addresses the gap of scalable, design-compliant feedback in large-scale teaching scenarios.
- **A Static Analysis-Driven Pipeline and High-Fidelity Dataset:** We construct a specialized high-fidelity fine-tuning dataset tailored for OOD courses, leveraging a static analysis-driven pipeline. This pipeline filters out implementation-level noise from raw student code, captures the architectural evolution of iterative course projects, and provides a high-quality, domain-specific foundation for the Helper framework's fine-tuning process.
- **An empirical Validation in a Real-World OOD Course:** We conduct comprehensive empirical evaluations of the Helper framework in a real undergraduate OOD course. Quantitative results demonstrate OO-Helper outperforms general foundation models in refactoring recommendation accuracy, while qualitative feedback from students and instructors validates its effectiveness as a pedagogical scaffold.

II. Related Work

Automated code review and refactoring have evolved through three approaches [7]. First, rule-based static analysis checks compliance but often lacks semantic depth for flexible object-oriented design [8], [9]. Second, retrieval-based methods use review corpora but depend heavily on dataset size [10], [11]. Third, model-based approaches, particularly those using Deep Learning and LLMs, map code to comments with increasing accuracy via fine-tuning and hybrid "identification + verification" architectures [12]–[15]. Concurrently, LLMs have demonstrated strong code generation capabilities. While early models struggled with object-oriented tasks [16], [17], recent advancements in prompt engineering and model iteration have improved performance in reducing code smells and adhering to style guidelines [18], [19].

In the educational domain, LLMs function as resource generators [20] and intelligent tutors for explanation and debugging [21]. Current research predominantly focuses on functional correctness, using LLMs to fix syntactic errors or repair logic [22]. However, a significant gap remains in guiding students to write good code rather than just working code [23]–[27]. Existing tools often overlook code quality metrics like logical structure and coupling, relying on direct correction rather than Socratic guidance that encourages student reflection and proactive quality improvement [28]–[31].

Despite these advancements, critical gaps persist in undergraduate object-oriented education: tools often overlook the incremental requirements nature of student projects, lack flexible evaluation of design principles, and fail to provide scalable, personalized architectural guidance. Addressing these needs, this study proposes a design optimization technique integrating static analysis with LoRA-fine-tuned LLMs within an Object-Oriented Design course. Unlike general-purpose tools, our approach automates personalized architectural recommendations, leveraging LLMs to cultivate higher-order programming skills by balancing code understanding with educational adaptability.

III. Methodology

A. Overview of Helper Framework

As illustrated in Figure 1, this pipeline transforms raw educational data into an intelligent assistant that guides iterative software architecture design. The proposed Helper framework consists of three distinct phases: (1) Fine-Tuning Data Preparation. Aligning with the data construction methodology detailed in Section III-B, this phase represents the preparation of the finalized dataset for the model. (2) Parameter Efficient Fine-Tuning. We utilize Lora [32] to balance training efficiency and model performance. (3) Suggestion Generation. In the inference stage, Helper serves as an interactive assistant for students. It analyzes these inputs to comprehend the architectural impact of the new demands and then generates a structured set of iterative suggestions. This pipeline transforms raw educational data into an intelligent assistant that guides iterative software architecture design.

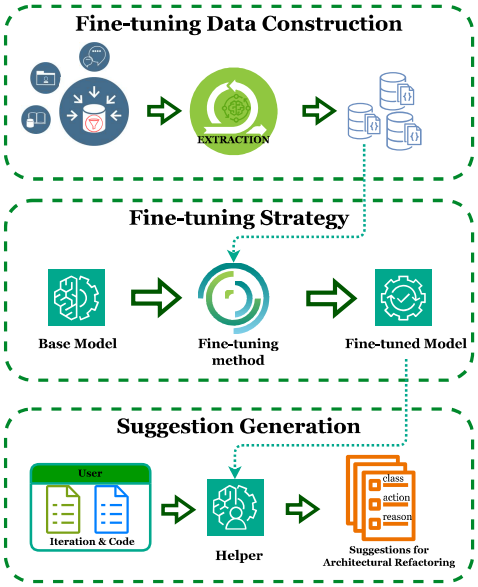

Fig. 1. The overall architecture of the Helper framework comprises fine-tuning data construction, fine-tuning strategy, and suggestion generation.

B. Construction of the Fine-tuning Dataset

The fine-tuning dataset is constructed through a rigorous cascading selection process: first, filtering by data completeness; second, selecting by architectural quality criteria using quantitative thresholds to enforce design principles and eliminate technical debt; third, selecting by functional correctness via retaining only the top scored submissions; and finally, raw code is abstracted into a specific format to focus on structural relationships and iteration evolution.

Based on the above general framework, we implement data preparation for the OOD course, as illustrated in Figure 2, with detailed operation steps as follows:

1) Data Collection: The data collection phase adopts training units from teaching practice (each spanning three teaching weeks and consisting of three project assignments) as the basic unit. Raw task-related data are collected from three sources: the experimental platform of the Object-Oriented Design course, students' project code-hosting platform, and course project blogs. A preliminary screening is conducted to ensure the completeness of students' assignment data. In this study, a total of 1,508 student datasets from 6 training units (Units 1, 2, and 4 of the 2023 Spring Semester and 2024 Spring Semester of the Object-Oriented Design course) were collected, along with students' summary blog data published for each unit. Each training unit includes three assignments. The statistics of raw data for each training unit are presented in Table I and Figure 3.

2) Data Filtering: After excluding datasets lacking code or blog content and screening all valid data, 1,177 raw student projects code and reflection blogs were obtained. On this basis, we perform further screening of the samples in accordance

TABLE I
STATISTICS OF COLLECTED PROJECT CODE BY UNIT

Unit	Samples	Score		Class Count	
		Mean	Std	Mean	Max
2023 Unit 1	649	74.1	24.5	11.9	52
2023 Unit 2	666	72.1	29.4	7.7	24
2023 Unit 4	654	86.6	25.6	12.1	27
2024 Unit 1	759	84.6	24.2	13.1	58
2024 Unit 2	779	81.4	26.3	9.5	29
2024 Unit 4	782	93.3	22.1	8.4	29
Total/Avg	**4289**	**82.4**	26.4	**10.4**	58

with two core SOLID principles: the Single Responsibility Principle (SRP) and the Open/Closed Principle (OCP).

To strictly enforce the SRP and prevent the accumulation of technical debt, we established quantitative constraints on code complexity and cohesiveness. Adhering to the OOD course's curriculum standards, we set a maximum of 500 lines per class (LOC Class) and 80 lines per method (LOC Method). We also evaluated method cohesiveness using LCOM, as defined by Sharma *et al.* [33]. To model method-attribute interactions, we applied a cutoff threshold of 0.7, as higher values indicate lower cohesion and potential SRP violations. Furthermore, regarding control flow complexity, while a Cyclomatic Complexity (CC) of 10 is conventionally considered the limit, we relaxed the method-level threshold to 20 to ensure data sufficiency, imposing no constraints on structurally trivial methods that are irrelevant to architectural complexity.

To facilitate the OCP and ensure maintainable extensibility, we focused on inheritance hierarchies and overall structural integrity. Excessive inheritance depth deteriorates readability, so we established a maximum threshold of 4 for the Depth of Inheritance Tree. Finally, as assignments with excessively low manual evaluation scores cannot guarantee functional correctness or structural validity, we selected the top 80% of data based on the score distribution of each training unit to

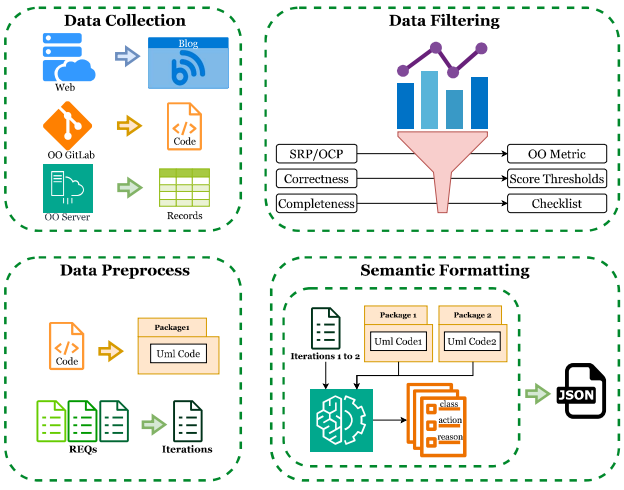

Fig. 2. The fine-tuning dataset construction pipeline consists of four parts.

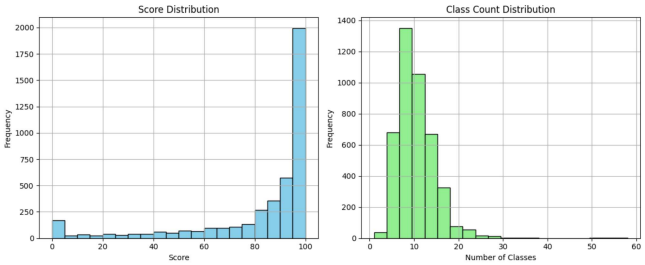

Fig. 3. This is the raw data distribution of score and class count.

TABLE II
STATISTICS OF THE FINE-TUNING DATASET: SOURCE DISTRIBUTION AND TRAIN/TEST SPLIT DETAILS

Panel A:Distribution of Fine-tuning Samples by Unit	
Unit	**Count**
2023 Unit 1	106
2023 Unit 2	168
2023 Unit 4	198
2024 Unit 1	242
2024 Unit 2	232
2024 Unit 4	358
Total Samples	**1304**

Panel B: Final Dataset Statistics (Train vs. Test)		
Metric	**Train Set**	**Test Set**
Samples Count	1,180	124
Total Actions	10,651	1,119
Avg. Actions/Sample	9.03	9.02
Action Category	**Count (Percentage)**	**Count (Percentage)**
add_class	2,985 (28.03%)	310 (27.70%)
remove_class	267 (2.51%)	19 (1.70%)
add_method	3,606 (33.86%)	372 (33.24%)
remove_method	175 (1.64%)	13 (1.16%)
modify_method	1,644 (15.44%)	208 (18.59%)
others	1,974 (18.53%)	197 (17.61%)

Panel C: Scale & Token Statistics		
Metric (Avg. ± Std.)	**Train Set**	**Test Set**
Input Length (Chars)	6,705 ± 2,337	6,874± 2,245
Output Length (Chars)	1,009 ± 223	999 ± 199
Input Tokens	2989 ± 653	3,043 ± 639
Output Tokens	584 ± 148	577 ± 142
Total Tokens	**3,573±655**	**3,621±630**
Max Total Tokens	7,517	5,648

ensure the dataset meets these architectural quality standards.

After screening using the indicators mentioned above, this paper ultimately obtained 654 data groups.

3) Data Preprocess: After excluding sensitive and irrelevant information, our data preprocessing includes the following two steps: extracting requirements from the assignment prompts and reversely generating PlantUML class diagram text from the original Java code.

In each iteration of the assignments, we extract the original and new requirements encompassing the model scenario, functional requirements, and functional data constraints. The

raw data comprises student assignment samples from six units, with three coding assignments per unit that generate two sets of task iteration data, resulting in a total of 12 iterations.

Direct input of raw code is suboptimal due to excessive implementation details. To address this, we utilize a javalang-based static analyzer to convert source code into PlantUML diagrams. This conversion serves two critical functions: it abstracts away syntactic noise to focus on structural relationships and explicitly maps scattered dependencies to aid architectural reasoning. In the process of static analysis, we parse the abstract syntax tree to extract only class signatures and their interrelationships, ensuring the model receives a holistic view of the system design without interference from local implementation logic.

4) Semantic Formatting: Finally, for the selected high-quality samples, this paper adopted DeepSeek-R1-671B for semantic formatting. For the convenience of observation and processing, system prompts are used to process modification operations and reason data into a JSON object array for output. Each JSON object includes modification location in the "class" field, operation type in the "action" field, and operation reason in the "reason" field.

With compliance to the predefined model input token limit, we extracted 1,304 pieces of iterative modification data from 654 groups of data. The distribution of the extracted data is shown in Table II.

C. Fine-tuning Large Language Models for Personalized Code Architecture Optimization

The fine-tuning implementation module aims to adapt pre-trained LLMs to the refactoring recommendation task in software engineering education. For the OOD course scenario in software engineering education, this paper conducts LoRA fine-tuning on LLMs such as Qwen2.5 and Qwen3, and adopts a two-stage structure for OOD suggestion refinement.

1) Fine-tuning Process: The fine-tuning dataset is structured in accordance with the platform's specifications, with its format presented in Figure 4.

First, LoRA models the weight update matrix $\Delta \boldsymbol{W}$ by imposing a low-rank constraint as follows:

$$\Delta \boldsymbol{W} = \frac{\alpha}{r} \cdot \boldsymbol{A}\boldsymbol{B}, \tag{1}$$

where $\boldsymbol{A} \in \mathbb{R}^{d \times r}$ and $\boldsymbol{B} \in \mathbb{R}^{r \times k}$ are trainable low-rank matrices, r is the rank ($r \ll \min(d, k)$), and α is a constant scaling hyperparameter. We use this decomposition to approximate the weight changes, which significantly reduces the number of trainable parameters compared to full-rank tuning.

Subsequently, LoRA formulates the forward propagation process to integrate the frozen pre-trained weights with the learned adaptation:

$$\text{Output} = \boldsymbol{h} \cdot \left(\boldsymbol{W_0} + \frac{\alpha}{r} \boldsymbol{A}\boldsymbol{B} \right), \tag{2}$$

where $\boldsymbol{h}$ represents the input representation and $\boldsymbol{W_0}$ denotes the frozen pre-trained weight matrix. We add the adapter branch to the original pathway, which ensures that the model retains its general linguistic capabilities while adapting to the specific architectural domain.

Training Sample Structure (SFT Format)

System Instruction

You are an assistant for PlantUML class diagram iteration...
Input Format: `<originReq>...<uml>...<newReq>...`
Output Format: JSON Array.

User Input

```
<originReq> System needs to support ... </originReq>
<uml> class Expression { ... } </uml>
<newReq> Add support for sin/cos factors... </newReq>
```

Assistant Output (Ground Truth)

```
[
  {
    "class": "Tri",
    "action": "Add Class",
    "reason": "Add generic container for sin/
    cos parsing..."
  }
]
```

Fig. 4. This is an example of constructed instruction tuning data.

Finally, LoRA defines the optimization rule where gradients are back-propagated exclusively to the adapter matrices:

$$\begin{cases} \boldsymbol{A}_{\text{new}} = \boldsymbol{A} - \eta \cdot \nabla_{\boldsymbol{A}} \mathcal{L}_{\text{Total}} \\ \boldsymbol{B}_{\text{new}} = \boldsymbol{B} - \eta \cdot \nabla_{\boldsymbol{B}} \mathcal{L}_{\text{Total}}, \end{cases} \tag{3}$$

where η is the learning rate and $\mathcal{L}_{\text{Total}}$ is the objective loss function. We update only the low-rank matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ while keeping $\boldsymbol{W_0}$ static, which significantly improves computational efficiency and memory utilization during training.

Each piece of data contains three types of information: system instruction, user input((original requirements, original PlantUML class diagram text, and new requirements), and assistant output(details of iterative modification operations and their reasons). In this paper, 90% of the data from each training unit—amounting to 1180 samples—was randomly selected as the training set.

2) Design of Refinement Task: This paper adopts a two-stage structure for OOD suggestions Refinement. In the first stage, the fine-tuned Qwen2.5-14B model is used to generate initial Advice. In the second stage, Qwen3-14B is employed to refine the iterative suggestions, merge similar items, and generate the final results. All suggestions generated follow the JSON array format of "class-action-reason." In the second stage, the model refines and merges the items where the action is "modify method". To accomplish the refinement and merging tasks, this paper adopts a task decomposition prompt method. The task decomposition process is illustrated in the following Figure 5.

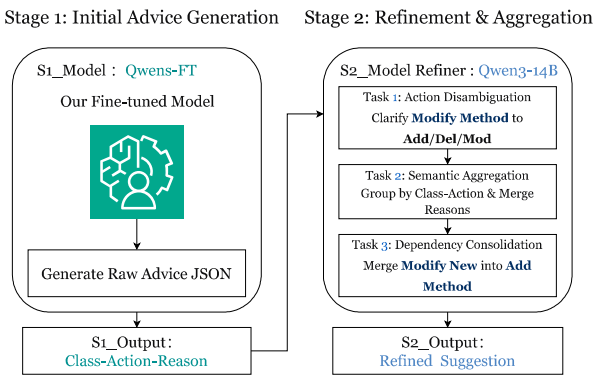

Fig. 5. This is OO-Helper's Two-Stage Workflow.

D. Suggestion Generation

In the suggestion generation phase, students submit their current code implementation alongside iterative requirements to the Helper framework. The system first executes static analysis on the submitted code to extract core artifacts that reflect high-level architectural evolution. Subsequently, these structured insights are fed into a fine-tuned LLM via prompt engineering to generate personalized refactoring recommendations tailored to the specific characteristics of the student's code. Specifically, within the OOD context, OO-Helper converts architectural data extracted through static analysis into UML text format. This representation is then synthesized with core OOD pedagogical principles and the student's iterative requirements into a unified prompt. Finally, this prompt is processed by the LoRA fine-tuned model to yield targeted iterative suggestions structured in a "class-action-reason" format.

E. Evaluation Metrics

To comprehensively assess the framework in the OOD course scenario, we employ a hybrid evaluation strategy combining quantitative benchmarks with qualitative expert review.

1) Precision and Recall: We adopt precision and recall to evaluate the accuracy of the model's refactoring recommendations against student ground truth. These metrics are standard in refactoring research [34], [35].

2) Human Evaluation: Since automated metrics often fail to capture architectural rationality, we introduce a human-evaluation mechanism based on a 5-point Likert Scale. We assess the suggestions across three core dimensions:

- **Suggestion Quality:** Evaluates the logical correctness of the code changes, clarity of the explanation, and alignment with new requirements.
- **Architectural Compliance:** Verifies adherence to fundamental design principles (SRP and OCP).
- **Practical Utility:** Measures the suggestion's real-world value, including the evaluator's willingness to adopt the code and the reduction in cognitive load.

To ensure reliability, final scores are calculated using a weighted mechanism based on the evaluator's proficiency with SOLID principles and the project context.

IV. Experiment

A. Automatic Evaluation results

To minimize generation randomness and ensure reproducibility of evaluation results, the temperature parameter was fixed at 0.1 across all LLM inference tasks. Table III presents a comparative evaluation of our fine-tuned models against their base models across five distinct refactoring categories. The results underscore the critical importance of domain-specific fine-tuning for structural code evolution tasks.

Table III shows that our fine-tuning strategy yields consistent gains across model sizes, with the greatest impact on Qwen3-14B. While the base model struggled with complex refactorings (16.93% recall in **modify_method**), Qwen3-14B-FT tripled this metric to 51.27% and boosted **add_class** precision from 30.36% to 55.33%. This turnaround indicates that our dataset effectively activates the model's latent potential for high-level architectural reasoning. This robustness is replicated across other architectures. For Qwen2.5-14B, fine-tuning bridged the capability gap, elevating **add_class** recall from a negligible 2.60% to 29.74%. Conversely, for the stronger Qwen3-32B baseline, fine-tuning refined decision boundaries doubled performance on high-risk tasks like **remove_method** (15.00% to 35.00%). Collectively, these results substantiate the value of our OOD-specific dataset in aligning general-purpose LLMs with educational contexts. The consistent improvements confirm that these gains stem from high-quality instruction tuning rather than model-specific artifacts. By injecting pedagogical rules, our dataset successfully shifts the focus from mere syntactic correctness to structural evolution, demonstrating that carefully curated data can significantly amplify the architectural intelligence of foundation models.

B. Human Evaluation

We invited 44 students who used OO-Helper to complete a questionnaire* designed to assess both the tool's practical utility and its trustworthiness in an educational setting. To gauge user receptiveness, we adapted metrics from the Technology Acceptance Model [36], specifically measuring Adoption Willingness and Perceived Efficiency. Simultaneously, following intelligent tutoring methodologies [37], we evaluated Architectural Accuracy and Alignment Strategy to verify the system's reliability with respect to hallucinations and adherence to requirements. To quantify the reliability of the assessment, we first analyzed the technical background of the participants. Statistics show that the average self-assessment score of evaluators regarding their understanding of design principles and domain requirements was 3.98/5.0.

The statistical results of the core metrics are presented in Table IV. Comprehensive analysis indicates that the model

*https://github.com/LiZuchen/OO-Helper-Supplementary/blob/main/questionnaire.pdf

TABLE III
THIS IS THE COMPARATIVE RESULTS OF REFACTORING RECOMMENDATION PERFORMANCE ON DIFFERENT LLMS. **BOLD** INDICATES THE BEST PERFORMANCE FOR EACH METRIC IN THE CORRESPONDING CATEGORY.

Category	Qwen2.5-14B		Qwen2.5-14B-FT		Qwen3-14B		Qwen3-14B-FT		Qwen3-32B		Qwen3-32B-FT	
	Recall	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**
`add_class`	2.60%	**68.18%**	**29.74%**	51.96%	30.94%	30.36%	**33.00%**	**55.33%**	**37.14%**	46.76%	30.83%	**55.21%**
`remove_class`	0.00%	0.00%	**23.08%**	**50.00%**	0.00%	0.00%	**7.69%**	**7.14%**	0.00%	0.00%	**11.54%**	**11.11%**
`add_method`	66.43%	59.26%	**81.05%**	**59.58%**	58.35%	50.51%	**66.18%**	**64.63%**	**68.70%**	56.14%	65.40%	**64.02%**
`remove_method`	**10.00%**	**15.38%**	0.00%	0.00%	5.00%	**20.00%**	**20.00%**	15.38%	15.00%	**37.50%**	**35.00%**	35.00%
`modify_method`	**64.39%**	45.31%	54.57%	**48.38%**	16.93%	**43.21%**	**51.27%**	36.93%	**68.72%**	44.17%	59.57%	**48.25%**

TABLE IV
THIS IS THE STATISTICAL RESULTS OF THE HUMAN EVALUATION CORE METRICS.

Dimension	Metric	Result
Architectural Accuracy (Mean Score / 5.0)	Design Principle	4.27
	Nec. of Class Mod	4.50
	Nec. of Action Mod	4.43
	Nec. of Reason Mod	4.40
Utility & Efficiency (Mean Score / 5.0)	Adoption Willingness	4.33
	Architectural Confidence	4.20
	Development Efficiency	3.83
Alignment Strategy (Distribution %)	Perfect Mapping	66.7%
	Recall Gap	23.3%
	Suggestion Redundancy	6.7%
	Dual Defects	3.3%

excels in the accuracy and standardization of architectural decisions, though there is room for improvement in the recall rate of long-tail requirements. We conduct a detailed analysis of the following three aspects.

1) Architectural Accuracy: The model achieved the highest score in the Architectural Accuracy dimension ($Mean = 4.44$). In the sub-metrics, the score for "Necessity of Class Modification/Addition" reached 4.50. This result significantly indicates that the model is capable of precisely identifying entity boundary changes triggered by requirement shifts. It effectively judges when to respond to requirements by extending new classes rather than modifying old ones, thereby avoiding the creation of "God Classes".

Furthermore, the Design Principle (SRP/OCP) Compliance score reached 4.27. Evaluators generally agreed that the suggestions generated by the model were not only functionally correct but also demonstrated expert-level capability in code decoupling. This confirms that the model has surpassed the scope of basic code completion and possesses the semantic understanding required to maintain a Clean Architecture.

2) Utility Analysis: Reduction of Cognitive Load: In terms of practical utility, the Adoption Willingness score reached ($Mean = 4.33$), demonstrating the high deployment value of the model's suggestions. Notably, students' understanding of development efficiency involves multiple complex factors, such as code quality, rather than being a single indicator of homework completion speed. It is important to emphasize that OO-Helper does not directly help students complete their homework; instead, it focuses on providing compliant architectural suggestions and decision support at the design level. Against this backdrop, in the matrix analysis of efficiency improvement, the score for Improvement in Architectural Confidence ($Mean = 4.20$) was significantly higher than that for mere Development Efficiency ($Mean = 3.83$).

This disparity reveals the core value logic of the tool: when facing complex iterative tasks, the suggestions provided by the model serve as a reliable "Second Opinion." By offering standardized architectural blueprints, it effectively reduces the developer's Cognitive Load and decision uncertainty, rather than merely shortening the physical time spent on code entry.

3) Alignment: High Precision vs. Hallucination: In the analysis of requirement alignment, the model exhibited a conservative but precise strategy.

- **Perfect Mapping: 66.7%** of the suggestions achieved a perfect mapping between requirements and code, while the proportion marked as Redundant Suggestions was only **6.7%**. This indicates that the model demonstrates extremely high Precision during generation, rarely producing redundant code that deviates from business logic.
- **Recall Gap:** Conversely, **23.3%** of the samples showed cases where "Requirements could not be fully met." Qualitative feedback analysis revealed that the model tends to overlook fine-grained edge rules when processing long-text requirements containing multiple constraints. This suggests that the current performance bottleneck lies mainly in the Recall of requirement capture.

V. CONCLUSION

In this paper, we present Helper, a modular LLM-based framework that provides automated, high-level architectural feedback. Through the construction of a specialized OOD dataset and the implementation of OO-Helper, we successfully bridged the gap between abstract design principles and concrete code implementation. Empirical validation confirms that our approach significantly surpasses general foundation models in refactoring accuracy and effectively fosters students' engineering skills. With its adaptable data pipeline and lightweight fine-tuning mechanism, the framework offers a practical, scalable solution for broad application across software engineering curricula.

ACKNOWLEDGMENT

This work was supported in part by the Engineering Research Center of Integration and Application of Digital Learning Technology, Ministry of Education under Grant 1431005, and the National Natural Science Foundation of China under Grant 62477001.

REFERENCES

[1] C. Hundhausen, P. Conrad, A. Tariq, S. Pugal, and B. Z. Flores, "An empirical study of the content and quality of sprint retrospectives in undergraduate team software projects," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, 2024, pp. 104–114.

[2] A. Williams, "Delivering effective student feedback in higher education: An evaluation of the challenges and best practice." *International Journal of Research in Education and Science*, vol. 10, no. 2, pp. 473–501, 2024.

[3] M. Messer, N. C. Brown, M. Kölling, and M. Shi, "Automated grading and feedback tools for programming education: A systematic review," *ACM Transactions on Computing Education*, vol. 24, no. 1, pp. 1–43, 2024.

[4] T. Phung, V.-A. Pădurean, J. Cambronero, S. Gulwani, T. Kohn, R. Majumdar, A. Singla, and G. Soares, "Generative ai for programming education: benchmarking chatgpt, gpt-4, and human tutors," in *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 2*, 2023, pp. 41–42.

[5] B. P. Cipriano and P. Alves, "Llms still can't avoid instanceof: An investigation into gpt-3.5, gpt-4 and bard's capacity to handle object-oriented programming assignments," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, 2024, pp. 162–169.

[6] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.

[7] U. Yadav and U. Shrawankar, "Artificial intelligence across industries: A comprehensive review with a focus on education," *AI applications and strategies in teacher education*, pp. 275–320, 2025.

[8] N. Palvannan and C. Brown, "Suggestion bot: analyzing the impact of automated suggested changes on code reviews," in *2023 IEEE/ACM 5th International Workshop on Bots in Software Engineering (BotSE)*. IEEE, 2023, pp. 33–37.

[9] O. Hazzan and Y. Erez, "Generative ai in computer science education," in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*, 2024, pp. 1899–1899.

[10] Y. Hong, C. Tantithamthavorn, P. Thongtanunam, and A. Aleti, "Commentfinder: a simpler, faster, more accurate code review comments recommendation," in *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*, 2022, pp. 507–519.

[11] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using pre-trained models to boost code review automation," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 2291–2302.

[12] I. Jaoua, O. B. Sghaier, and H. Sahraoui, "Combining large language models with static analyzers for code review generation," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 2025, pp. 174–186.

[13] M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui, "Exploring parameter-efficient fine-tuning techniques for code generation with large language models," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 7, pp. 1–25, 2025.

[14] H. Feng, P. Zhao, Q. Sun, C. Xu, F. Yang, L. Wang, Q. Ma, Q. Lin, S. Rajmohan, D. Zhang *et al.*, "Warriorcoder: Learning from expert battles to augment code large language models," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2025, pp. 4955–4969.

[15] R. Luo, H. Zhang, L. Chen, T.-E. Lin, X. Liu, Y. Wu, M. Yang, Y. Li, M. Wang, P. Zeng *et al.*, "Mmevol: Empowering multimodal large language models with evol-instruct," in *Findings of the Association for Computational Linguistics: ACL 2025*, 2025, pp. 19 655–19 682.

[16] J. Finnie-Ansley, P. Denny, A. Luxton-Reilly, E. A. Santos, J. Prather, and B. A. Becker, "My ai wants to know if this will be on the exam: Testing openai's codex on cs2 programming exercises," in *Proceedings of the 25th Australasian Computing Education Conference*, 2023, pp. 97–104.

[17] B. P. Cipriano and P. Alves, "Gpt-3 vs object oriented programming assignments: An experience report," in *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, 2023, pp. 61–67.

[18] J. Cordeiro, S. Noei, and Y. Zou, "An empirical study on the code refactoring capability of large language models," *arXiv preprint arXiv:2411.02320*, 2024.

[19] A. Midolo and M. Di Penta, "Automated refactoring of non-idiomatic python code: A differentiated replication with llms," *arXiv preprint arXiv:2501.17024*, 2025.

[20] U. Mittal, S. Sai, V. Chamola, and D. Sangwan, "A comprehensive review on generative ai for education," *Ieee Access*, vol. 12, pp. 142 733–142 759, 2024.

[21] M. Liffiton, B. E. Sheese, J. Savelka, and P. Denny, "Codehelp: Using large language models with guardrails for scalable support in programming classes," in *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*, 2023, pp. 1–11.

[22] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025, pp. 694–694.

[23] A. F. Pereira and R. F. Mello, "A systematic literature review on large language models applications in computer programming teaching evaluation process," *IEEE Access*, 2025.

[24] D. Franklin, P. Denny, D. A. Gonzalez-Maldonado, and M. Tran, *Generative AI in computer science education: Challenges and opportunities*. Cambridge University Press, 2025.

[25] N. Raihan, M. L. Siddiq, J. C. Santos, and M. Zampieri, "Large language models in computer science education: A systematic literature review," in *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*, 2025, pp. 938–944.

[26] K. K. C. Cheung, Y. Long, Q. Liu, and H.-Y. Chan, "Unpacking epistemic insights of artificial intelligence (ai) in science education: A systematic review," *Science & Education*, vol. 34, no. 2, pp. 747–777, 2025.

[27] A. Gaitantzi and I. Kazanidis, "The role of artificial intelligence in computer science education: A systematic review with a focus on database instruction," *Applied Sciences*, vol. 15, no. 7, p. 3960, 2025.

[28] I. Hilliger, M. Pérez-Sanagustín, E. Villalobos, R. Ferreira, and C. Gonzalez, "Scaffolding learning scenarios with a socratic chatbot: Insights from educators," in *European Conference on Technology Enhanced Learning*. Springer, 2025, pp. 155–161.

[29] F. Bazouche, "The socratic questioning method in the age of artificial intelligence (ai) in higher education," in *EdMedia+ Innovate Learning*. Association for the Advancement of Computing in Education (AACE), 2025, pp. 38–46.

[30] B. Degen, "Resurrecting socrates in the age of ai: A study protocol for evaluating a socratic tutor to support research question development in higher education," *arXiv preprint arXiv:2504.06294*, 2025.

[31] A. Chen, H. Wu, Q. Xin, S. P. Reiss, and J. Xuan, "Studying and understanding the effectiveness and failures of conversational llm-based repair," in *2025 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE, 2025, pp. 56–59.

[32] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021. [Online]. Available: https://arxiv.org/abs/2106.09685

[33] T. Sharma and D. Spinellis, "Do we need improved code quality metrics?" *arXiv preprint arXiv:2012.12324*, 2020.

[34] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[35] 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.

[36] P. Silva, "Davis' technology acceptance model (tam)(1989)," *Information seeking behavior and technology adoption: Theories and trends*, pp. 205–219, 2015.

[37] J. Lin, L. Sha, Y. Li, D. Gasevic, and G. Chen, "Establishing trustworthy artificial intelligence in automated feedback," 2022.