E4R-Reviewer: Effective and Explainable Automated Code Review via End-to-End Reasoning-Guided Alignment

YIFEI LIU*, Institute of Software, Chinese Academy of Sciences, China
XIZHI HOU*, Institute of Software, Chinese Academy of Sciences, China
LI YANG†, Institute of Software, Chinese Academy of Sciences, China
HUAN LIU*, Institute of Software, Chinese Academy of Sciences, China
CHEN ZHU, University of Chinese Academy of Sciences, China
FENGJUN ZHANG, Institute of Software, Chinese Academy of Sciences, China
CHUN ZUO, Sinosoft Company Limited, China

Code review is a key practice for ensuring software quality and maintainability. Despite progress in Automated Code Review (ACR), existing methods face two core challenges: **(1) Isolated Task Modeling.** Current approaches often model and optimize subtasks in ACR independently, ignoring the inherent logical order and internal dependencies among them which affects the effectiveness of ACR. **(2) Lack of Explainability.** At the task level, the absence of explanatory information in review comments increases developers' cognitive load; at the model level, the black-box nature fundamentally undermines developer trust.

To address these challenges, we propose **E4R-Reviewer**, improving the **E**ffectiveness and **E**xplainability of ACR via **E**nd-to-**E**nd **R**easoning-guided alignment. *For effectiveness*, E4R-Reviewer unifies multiple fine-grained ACR subtasks into a single end-to-end reasoning process, enabling cross-task knowledge sharing and allowing the model to explicitly complete a reasoning chain of "quality estimation–issue localization–issue classification–issue description–fix suggestion–code refinement" in one generation. Meanwhile, we adopt a Group Relative Policy Optimization (GRPO)-based reinforcement-learning alignment, treating the reasoning steps as optimizable intermediate objectives. We design subtask-specific rewards and integrate them via curriculum-inspired, multi-stage reward fusion that follows the real-world review workflow. *For explainability*, E4R-Reviewer produces reasoning process and generates structured review results covering all fine-grained ACR subtasks, improving the transparency and explainability of the review results.

Extensive evaluations on public, real-world datasets demonstrate that E4R-Reviewer significantly outperforms existing methods and achieves state-of-the-art performance: a 74.61% F1-score in quality estimation and +22.96% CodeBLEU in code refinement. Furthermore, Large Language Model (LLM) and human evaluation further confirm the superiority of E4R-Reviewer in terms of effectiveness and explainability.

CCS Concepts: • **Software and its engineering → Software creation and management**.

*Affiliated with University of Chinese Academy of Sciences, Beijing, China.
†Li Yang is the corresponding author.

Authors' Contact Information: Yifei Liu, Institute of Software, Chinese Academy of Sciences, China, liuyifei241@mails.ucas.ac.cn; Xizhi Hou, Institute of Software, Chinese Academy of Sciences, China, houxizhi2024@iscas.ac.cn; Li Yang, Institute of Software, Chinese Academy of Sciences, China, yangli2017@iscas.ac.cn; Huan Liu, Institute of Software, Chinese Academy of Sciences, China, liuhuan25@mails.ucas.ac.cn; Chen Zhu, University of Chinese Academy of Sciences, China, zhuchen226@mails.ucas.ac.cn; Fengjun Zhang, Institute of Software, Chinese Academy of Sciences, China, fengjun@iscas.ac.cn; Chun Zuo, Sinosoft Company Limited, China, zuochun@sinosoft.com.cn.

Conference'26, Oakland, California, United States

https://doi.org/XXXXXXX.XXXXXXX

Additional Key Words and Phrases: Automated Code Review, Large Language Models, Reinforcement Learning, Reasoning, Explainability

ACM Reference Format:
Yifei Liu, Xizhi Hou, Li Yang, Huan Liu, Chen Zhu, Fengjun Zhang, and Chun Zuo. 2026. E4R-Reviewer: Effective and Explainable Automated Code Review via End-to-End Reasoning-Guided Alignment. In *Proceedings of Conference'26.* ACM, New York, NY, USA, 25 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Code review is a cornerstone practice in modern software engineering that helps identify defects, facilitate knowledge transfer, and improve software quality [2, 10, 48, 63]. Despite its essential role, traditional manual code review is constrained by time and cognitive resources, making it infeasible to identify all potential defects and prone to subjective biases due to reviewers' varying expertise [3, 45].

To alleviate this burden, significant research efforts have been devoted to Automated Code Review (ACR), spanning early Deep Learning (DL)–based approaches [34, 38, 68] to recent Large Language Model (LLM)–based methods [41, 50, 77], which have shown considerable promise. Despite their potential, however, current ACR methods still face two core challenges:

(1) Isolated Task Modeling. As shown in Fig. 2(a), code review is a multi-stage, interdependent process where developers and reviewers sequentially perform multiple subtasks. However, prior work either models subtasks in isolation [38, 41, 68] or focuses on a single task—such as quality estimation [15, 55, 75], comment generation [24, 34, 37], or code refinement [13, 18, 19]—ignoring the logical order and internal dependencies. This isolated modeling fails to leverage shared knowledge or propagate context aligned with human review flow, yielding suboptimal and inconsistent results that undermine ACR effectiveness.

As shown in Fig. 1, conventional ACR methods—trained only for review comments generation—and the Qwen-2.5-Coder-7B-Instruct model both miss a critical logic error: replacing `query.StartIndex -= query.Limit` ("previous page") with `query.StartIndex += query.Limit`. Qwen offers only a surface-level summary, overlooking the reversed navigation direction and wrongly endorsing the change. In contrast, E4R-Reviewer, trained jointly with code refinement, uses repair insights to retroactively assess semantic correctness, identifying the inverted direction and recommending reversion to subtraction. This highlights how isolated task modeling in ACR fails to capture cross-task dependencies essential for accurate review.

(2) Lack of Explainability. At the task level, the review comments generation task in ACR suffers from a lack of explanatory content in human-written comments, due to reviewers' time and effort constraints [74]. Consequently, ACR models trained on such comments as ground truth also fail to provide explanations. However, developers strongly prefer explanatory comments [6], as they reduce cognitive load and lead to more efficient development cycles with fewer defects [6, 43, 54]. At the model level, existing DL- and LLM-based ACR methods operate as black-box systems, producing only final outputs without revealing their reasoning [21], which compromises both trust and practical utility.

To address these challenges, we propose **E4R-Reviewer**, improving the **E**ffectiveness and **E**xplainability of ACR via **E**nd-to-**E**nd **R**easoning-guided alignment. Unlike prior paradigms that model subtasks in isolation, E4R-Reviewer formulates code review as an end-to-end reasoning process that follows the real-world review workflow: the model operates on the same code change and sequentially performs quality estimation, issue localization, issue classification, issue description, fix suggestion, and code refinement, thereby leveraging the logical order and internal dependencies among subtasks within a unified context. For training, we adopt a Group Relative Policy Optimization (GRPO)-based Reinforcement-Learning (RL) alignment that assigns separate

idx 5270

```
@@ -76,7 +83,9 @@
define([“layoutManager”, “loading”, “events”, "libraryBrowser", "imageLoader",
            return;
        }
-       query.StartIndex -= query.Limit;
+       if (userSettings.libraryPageSize() > 0) {
+           query.StartIndex += query.Limit;
+       }
```

Ground Truth: ```suggestion query.StartIndex -= query.Limit; ``` or ideally `query.StartIndex = Math.max(0, query.StartIndex - query.Limit)`"

T5-Reviewer: please put the @ here the PR as well.

CodeReviewer: I think it would be better to check `if (userSettings.libraryPageSize() > 0 && query.StartIndex >= 0)`

LLaMA-Reivewer: I think this is a bug.

Qwen-2.5-Coder-7B finetuned on Review Comments Generation Task: Change the line `query.StartIndex -= query.Limit;` to `if (userSettings.libraryPageSize() > 0) { query.StartIndex += query.Limit; }` to ensure proper pagination behavior.

E4R-Reviewer (Only extract the parts related to review comments):
{“**issue_description**”: “The code has changed the operation performed on `query.StartIndex` from subtraction to addition. This change may have unintended consequences and should be reviewed to ensure the correct behavior is achieved.”, “**fix_suggestion**”: “Consider reverting the change and using the original subtraction operation.”}

Fig. 1. The Motivation Case of Isolated Task Modeling in ACR

reward functions to different subtasks and turns reasoning steps in the review chain into optimizable intermediate objectives. Since optimizing an end-to-end review objective over multiple interdependent subtasks can suffer from learning difficulties caused by a complex optimization target, we further introduce a curriculum-inspired, multi-stage reward fusion mechanism aligned with the real-world review workflow. This mechanism dynamically re-weights subtask rewards across training stages, enabling the model to progress from easier, prerequisite review capabilities to more complex reviewing and refinement abilities. To enhance explainability and usability, E4R-Reviewer explicitly outputs intermediate reasoning process and generates structured review results covering all fine-grained ACR subtasks, helping developers quickly locate key information, understand the underlying rationale, and adopt suggested fixes.

To validate E4R-Reviewer, we conduct an extensive experimental evaluation. Results show that E4R-Reviewer achieves state-of-the-art performance, with 73.60% accuracy, 74.61% F1-score, and +22.96% CodeBLEU, significantly outperforming all baselines. Moreover, considering the subjectivity of the generated outputs, comprehensive LLM and human evaluation is conducted. The results consistently confirm that E4R-Reviewer not only generates more effective review results but also provides richer explanatory information, thereby enhancing the explainability of the review process.

In summary, the main contributions of this paper are as follows:

- We propose E4R-Reviewer, which unifies multiple fine-grained ACR subtasks through end-to-end reasoning-guided alignment, thereby improving the overall effectiveness and explainability of ACR.
- We adopt a GRPO-based RL alignment and design a curriculum-inspired, multi-stage reward fusion mechanism to dynamically adjust the optimization emphasis across subtasks, enabling progressive learning of end-to-end reviewing capabilities.
- We enhance model explainability by producing structured review outputs with intermediate reasoning, and conduct extensive evaluations on real-world datasets, including quantitative

metrics as well as LLM and human evaluation. The results show that E4R-Reviewer consistently outperforms existing methods.

The rest of the paper is structured as follows: Sec. 2 presents a brief overview of the background. Sec. 3 elaborates on the various components of our proposed approach. Sec. 4 and Sec. 5 provide details of the experimental setup and experimental results. Sec. 6 reviews the related work. Sec. 7 illustrates potential threats to validity. Finally, Sec. 8 concludes the paper.

2 Background

2.1 Logical Order and Internal Dependencies in Automated Code Review

As shown in Fig. 2(a), code review is a multi-stage and interdependent process in which developers and reviewers sequentially complete a series of subtasks. In contrast, most conventional ACR methods decompose this workflow into isolated subtasks and train separate models (or objectives) using only subtask-specific data (Fig. 2(b)). Such isolated task modeling fails to leverage cross-stage contextual information and the dependencies between intermediate decisions, which often compromises both the effectiveness and the consistency of the outputs. We argue that an effective ACR model should capture the logical order of the review workflow and the internal dependencies among subtasks; otherwise, it may produce inconsistent conclusions across stages or converge to suboptimal solutions.

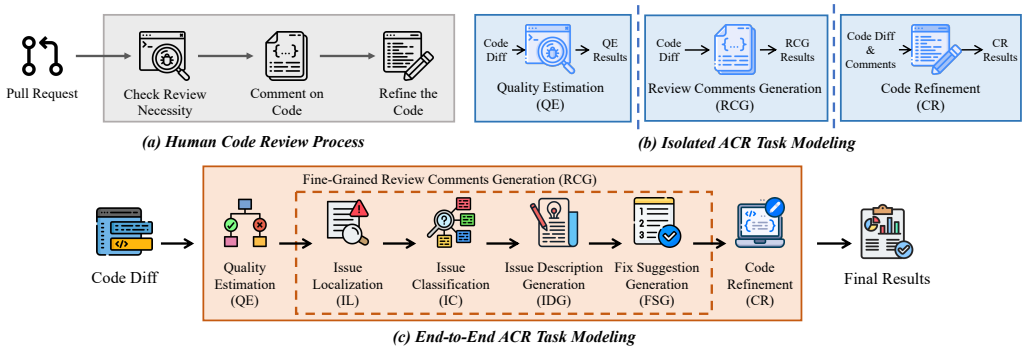

Fig. 2. Different task modeling approaches in ACR

(1) Logical Order: ACR subtasks follow a conditional sequence reflecting human cognition: reviewers first assess code quality (Quality Estimation, QE); if issues exist, they generate review comments (Review Comment Generation, RCG); developers then refine the code (Code Refinement, CR). This cascading flow should be explicitly modeled to mimic real-world review behavior.

(2) Internal Dependency: Subtasks mutually inform one another. Specifically, QE benefits from RCG/CR insights to better define quality issues. RCG relies on QE to decide whether to comment and uses CR's repair needs to generate actionable suggestions. CR, in turn, depends on RCG's structured feedback for precise fixes.

Recent studies have attempted to leverage the relationships between subtasks to improve the performance of ACR models. DISCOREV [5] adopts a knowledge distillation approach, constructing dependency pairs (e.g., CR and QE), where the student model builds a loss function based on supervised labels output by the teacher model to achieve knowledge transfer between tasks. However, it relies on static, paired combinations and only uses supervised loss, failing to fully exploit the system-wide synergies. Moreover, it requires maintaining both teacher and student models simultaneously, increasing training complexity.

Unlike DISCOREV's static pairwise distillation, our approach unifies all subtasks into a single end-to-end reasoning chain, where each subtask's output constitutes a sequential step in a coherent logical flow (Fig. 2(c)), avoiding the limitations and complexity of manually designing inter-task interactions, boosting performance by modeling the intrinsic logical flow of code review.

2.2 Task and Model Explainability in Automated Code Review

Previous ACR methods have primarily focused on technical accuracy and effectiveness [7], often overlooking the critical dimension of explainability [74, 77].

(1) **Task Explainability**: Task explainability primarily concerns the RCG task: ACR-generated review comments should include not only specific fix suggestions but also explanatory information to help developers understand issue root causes and improvement benefits. Studies show that 85% of developers prefer such causal explanations [6], which reduce post-release defects by 30% [43], cut review iterations by 15% [6], shorten resolution times by 15%, and boost developer satisfaction by 12% [54]. However, an empirical analysis found that 46% of real-world comments lack any explanation, offering only modification suggestions [74]. This deficiency is also reflected in existing ACR datasets: many "ground-truth" comments are derived from similarly non-explanatory human feedback, which encourages models trained on these data to mimic the same pattern and generate uninformative, insufficiently explainable comments—making them harder for developers to understand and act on, and ultimately prolonging the review cycle.

(2) **Model Explainability**: Model explainability concerns the ACR model itself. Since existing DL- and LLM-based ACR methods typically employ black-box models lacking transparency, users struggle to understand the model's decision rationale, undermining trust. To address this, recent work enhances model transparency by generating reasoning chains before producing outputs. These chains progressively reveal the model's internal reasoning, enabling users to trace how conclusions are reached and thereby increasing trust in the model's predictions.

3 Approach

The overall pipeline of the E4R-Reviewer is shown in Fig. 3, which mainly consists of four stages: A. End-to-End Automated Code Review Definition (Sec. 3.1), B. Dataset Construction (Sec. 3.2), C. Supervised Fine-Tuning (Sec. 3.3), and D. GRPO-based Reinforcement Learning (Sec. 3.4).

3.1 End-to-End Automated Code Review Definition

As mentioned in Sec. 2.1, typical ACR approaches abstract the key stages of the code review process into three main tasks [38, 68]:

(1) Quality Estimation (QE): Estimate whether the code changes in a PR have quality issues and require review:

$$s = f_{\text{QE}}(D(C_0, C_1)) \text{ where } s \in \{0, 1\}, \tag{1}$$

where $f_{\text{QE}}(\cdot)$ denotes the QE model, C_0 and C_1 denote the original and modified code, respectively, and $D(C_0, C_1)$ represents the code diff as the input. The output is a binary label $s \in \{0, 1\}$ indicating the presence of quality issues.

(2) Review Comment Generation (RCG): Generate natural language review comments based on the code diff which has quality issues:

$$T = f_{\text{RCG}}(D(C_0, C_1)) \text{ conditioned on } f_{\text{QE}}(D(C_0, C_1)) = 1, \tag{2}$$

where $f_{\text{RCG}}(\cdot)$ denotes the RCG model, and T denotes the generated review comments in natural language. The function is only invoked when the QE task indicates that there are quality issues.

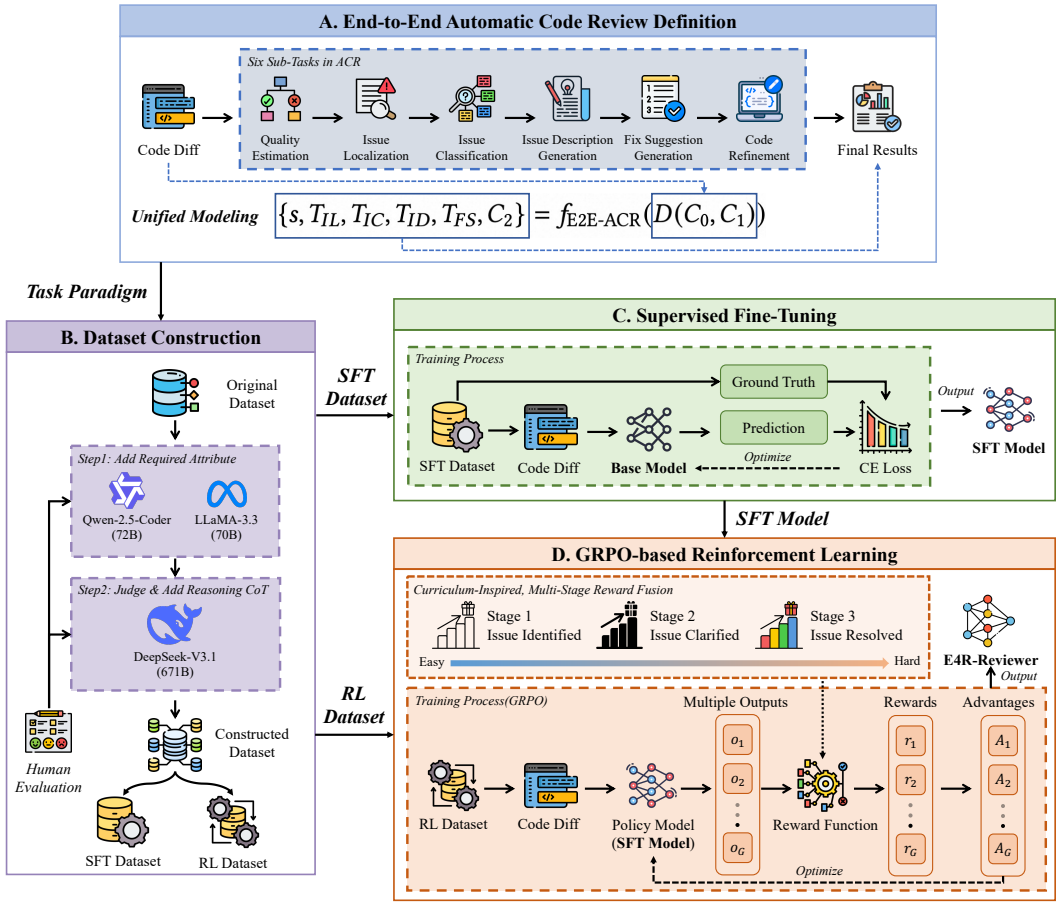

Fig. 3. The overall pipeline of E4R-Reviewer

(3) Code Refinement (CR): Refine the current code based on the review comments:

$$C_2 = f_{\text{CR}}(C_1, T), \tag{3}$$

where $f_{\text{CR}}(\cdot)$ denotes the CR model, and C_2 denotes the refined code after applying the review comments T to the modified code C_1.

However, the above task paradigm has many limitations which detailed in Sec. 2. To address these challenges, we first transform the raw, unstructured review comment T into a structured set $\{T_{IL}, T_{IC}, T_{ID}, T_{FS}\}$, corresponding to issue location, issue classification, issue description, and fix suggestion, respectively. Accordingly, the RCG task is decomposed into four sequential subtasks: Issue Localization (IL), Issue Classification (IC), Issue Description Generation (IDG), and Fix Suggestion Generation (FSG).

Furthermore, as we define an End-to-End Automated Code Review (E2E-ACR) paradigm that jointly models all review subtasks:

$$\{s, T_{IL}, T_{IC}, T_{ID}, T_{FS}, C_2\} = f_{\text{E2E-ACR}}(D(C_0, C_1)), \tag{4}$$

where $f_{\text{E2E-ACR}}(\cdot)$ denotes the E2E-ACR model that takes the raw code diff $D(C_0, C_1)$ as input, jointly models all subtasks of code review, including QE, IL, IC, IDG, FSG, and CR, and outputs the corresponding results: quality estimation label s, formatted review comments $\{T_{IL}, T_{IC}, T_{ID}, T_{FS}\}$, and refined code C_2.

Although subtasks loosely correspond to human roles—e.g., QE aligns more with reviewers and CR with developers—these boundaries are not rigid in ACR, as the system serves as an assistive tool for both parties. We therefore unify all subtasks within the E2E-ACR paradigm to facilitate cross-task information sharing and enhance performance.

3.2 Dataset Construction

3.2.1 Data Collection and Annotation. For our raw training dataset, we chose Carllm dataset [77] over the more commonly used CodeReviewer dataset [38] for three reasons: (1) CodeReviewer only provides coarse-grained subtasks, requiring extensive manual augmentation to support the missing subtasks; Carllm, by contrast, already structures comments into issue location, description, and fix suggestions—closely matching our task requirements. (2) CodeReviewer organizes data per isolated subtask, hindering end-to-end modeling, whereas Carllm integrates quality assessment with issue-level details, better supporting our unified pipeline. (3) Despite its popularity, CodeReviewer suffers from noisy, low-informativeness comments and excessive size; Carllm is smaller yet higher-quality, making it more suitable for RL.

Based on the Carllm dataset, we further added issue type and refined code fields to support the IC and CR tasks. First, we use two heterogeneous medium-scale LLMs—Qwen-2.5-Coder-72B and LLaMA-3.3-70B—feeding each the code diff to generate candidate answers, where model heterogeneity enhances diversity and robustness. Second, we employ the large-scale DeepSeek-V3.1 (671B) as a judge to evaluate these candidates and select the final answer, a process akin to 2-shot prompting that improves data quality. Additionally, we prompt DeepSeek-V3.1 to generate detailed reasoning chains that explicitly capture logical dependencies among subtasks. These chains serve as supervision signals in later training stages, enabling reasoning capability distillation into smaller models.

To ensure dataset quality, we conducted a manual evaluation on 370 randomly selected samples (providing 95% confidence level with ±5% margin of error). Two software engineering master's students—each with ≥4 years of software development and code review experience—independently assessed the added *issue type* and *refined code* fields via binary judgments, and rated the reasoning chains on correctness, thoroughness, and clarity using a 1–5 Likert scale. Results show >95% accuracy for the structured fields. The average reasoning scores in correctness/thoroughness/clarity are 4.78/4.83/4.76, and the corresponding Weighted Kappa [76] values are 0.90/0.93/0.87.

3.2.2 Training Data Format for SFT and GRPO. We use the same dataset for both SFT and GRPO. Below we describe the prompt template and output format used to construct each training sample.

As shown in Fig. 4, the prompt first provides task instructions and asks the model to perform step-by-step reasoning following the subtask order in the E2E-ACR pipeline. It then specifies a unified output format with two sections: `<reasoning>` and `<answer>`. The `<reasoning>` section contains the complete reasoning process to support explainability, while the `<answer>` section is a JSON object that contains the outputs for all subtasks. This structured design improves readability for developers and enables automatic parsing for downstream reward computation and metric evaluation.

During SFT, we reorganize the original dataset into the above supervision format to obtain ground-truth labels. During GRPO, we extract subtask outputs from sampled model responses and compute rewards using rule-based reward functions, requiring no additional data construction.

Instruct Prompt

You are a Senior Software Engineer performing a code review. \n The input is a code diff to review, analysis step by step:
(1) Quality Estimation \n **(2)** Issue Localization (locate the problematic code) \n **(3)** Issue Classification (Refactoring, Bugfix, Testing, Logging, Documentation, or Other) \n **(4)** Issue Description Generation \n **(5)** Fix Suggestion Generation \n **(6)** Code Refinement
Respond in the following format:

Target Output Format

```
<reasoning>
... detailed step-by-step review reasoning ...
</reasoning>
<answer>
{
  "has_issues": true/false,
  "issues": [{"issue_location": "...", "issue_type": "...", "issue_description": "...", "fix_suggestion": "..."}...],
  "refined_code": ""
}
</answer>
```

Note: Properly escape special characters in JSON strings (\\n, \\", \\, etc.).
Code to review: {code_diff}

Fig. 4. The prompt template and output format for SFT and GRPO training.

Rule-based rewards avoid the need for manually annotated preference pairs as in Direct Preference Optimization (DPO).

3.3 Supervised Fine-Tuning

Recent work on reasoning LLMs typically adopts a two-stage training paradigm: a SFT phase for cold start precedes RL. This stage enables the model to rapidly acquire the required output format and task structure, providing a stable foundation for RL. In our setting, the model must generate outputs for all subtasks in a strict JSON schema. Directly applying RL from the base model often fails in early stages, as the model struggles to produce valid JSON, hindering subtask answer extraction and reward computation, thereby degrading RL effectiveness.

In the SFT stage, we use the standard Cross-Entropy (CE) loss as the objective function:

$$\mathcal{L}_{SFT} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T_i}\log P(\mathbf{y}_{i,t}|\mathbf{x}_i, \mathbf{y}_{i,<t}; \Theta) \tag{5}$$

where N is the number of training samples, T_i is the length of the output sequence for sample i, $\mathbf{x}_i$ is the input code diff, $\mathbf{y}_i$ is the corresponding label (including reasoning process and answers for six subtasks), and Θ represents the model parameters.

Prior work has explored custom loss functions in SFT to better align with task objectives beyond standard CE loss. For instance, Carllm [77] isolates the quality estimation token and computes its loss separately to emphasize output quality for that component. In contrast, our SFT stage prioritizes learning the correct output format and task structure; thus, we adopt the standard CE loss without additional customizations.

3.4 GRPO-Based Reinforcement Learning

In RL stage, we adopt the Group Relative Policy Optimization (GRPO) algorithm [16, 58] to optimize the model.

3.4.1 Definition of GRPO. GRPO is a variant of PPO that eliminates the need for a separate critic model by computing group-relative advantages from the average reward of sampled outputs for the same input, significantly reducing computational overhead.

Formally, given a question q, GRPO samples a group of G outputs $\{o_1, o_2, \ldots, o_G\}$ from the old policy $\pi_{\theta_{\text{old}}}$. Let r_i be the scalar reward assigned to output o_i by a reward model. The normalized

(relative) reward for output o_i is computed as:

$$\tilde{r}_i = \frac{r_i - \mu_r}{\sigma_r}, \quad \text{where } \mu_r = \frac{1}{G}\sum_{j=1}^{G} r_j,\ \sigma_r = \sqrt{\frac{1}{G}\sum_{j=1}^{G}(r_j - \mu_r)^2} \tag{6}$$

For outcome supervision, the advantage $\hat{A}_{i,t}$ for every token t in output o_i is set to this normalized reward, i.e., $\hat{A}_{i,t} = \tilde{r}_i$.

The GRPO objective is then defined as:

$$J_{\text{GRPO}}(\theta) = \mathbb{E}_{q,\{o_i\}}\left[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|o_i|}\sum_{t=1}^{|o_i|}\left(\min\left[r_{i,t}^{\text{ratio}}\hat{A}_{i,t},\ \text{clip}(r_{i,t}^{\text{ratio}}, 1-\varepsilon, 1+\varepsilon)\hat{A}_{i,t}\right] - \beta\, D_{\text{KL}}[\pi_\theta \| \pi_{\text{ref}}]\right)\right] \tag{7}$$

where $r_{i,t}^{\text{ratio}} = \frac{\pi_\theta(o_{i,t}|q,o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q,o_{i,<t})}$ is the probability ratio between the current policy π_θ and the sampling policy $\pi_{\theta_{\text{old}}}$, ε is the PPO clipping hyperparameter, π_{ref} is a frozen reference policy (typically the SFT model), and $D_{\text{KL}}[\pi_\theta \| \pi_{\text{ref}}]$ is approximated per-token using an unbiased estimator.

3.4.2 Task-Specific Reward Function. The reward model plays a crucial role in RL training, as it provides feedback signals that guide the model to learn desired behaviors. While traditional RL approaches often rely on human preference-based reward models, recent advancements, such as DeepSeek-R1 [16], have demonstrated the effectiveness of rule-based reward functions for training reasoning LLMs. So we design a rule-based reward function which contains two main components: Format Reward R_f and Content Reward R_c. The overall reward R is formulated as:

$$R = \lambda_f R_f + \lambda_c R_c, \tag{8}$$

where λ_f and λ_c are hyperparameters that balance the contributions of the two reward components.

Format Reward R_f: The format reward evaluates whether the model's output adheres to the required structured format, which is defined as:

$$R_f = \begin{cases} 1.0, & \text{if } \hat{y} \text{ follows the XML-style schema and JSON structure with target keys,} \\ 0.5, & \text{if } \hat{y} \text{ follows the XML-style schema,} \\ 0.0, & \text{otherwise,} \end{cases} \tag{9}$$

where $\hat{y}$ is the model output, and the XML-style schema and JSON structure with target keys are defined in Sec. 3.2.2. Notably, although for non-issue samples the fields related to issues and refined code are not essentially required, we still instruct the model to output these fields (with empty string values) to maintain consistency in the format reward function design.

Content Reward R_c: The content reward assesses the correctness and quality of the model's output for each subtask. However, the IDG and FSG subtasks output natural language text, which is highly subjective and challenging to have a gold-standard reference for evaluation. Therefore, we only design content rewards for the QE, IL, IC, and CR subtasks, which can be defined as:

$$R_c = \lambda_{\text{QE}} R_{\text{QE}} + \lambda_{\text{IL}} R_{\text{IL}} + \lambda_{\text{IC}} R_{\text{IC}} + \lambda_{\text{CR}} R_{\text{CR}}, \tag{10}$$

where R_{QE}, R_{IL}, R_{IC}, and R_{CR} are the content rewards for the QE, IL, IC, and CR subtasks, respectively. λ_{QE}, λ_{IL}, λ_{IC}, and λ_{CR} are hyperparameters balancing the contributions of each subtask's content reward.

QE subtask Reward $R_{\mathbf{QE}}$: The QE subtask is a binary classification task, so we define its content reward as:

$$R_{\text{QE}} = \begin{cases} 1.0, & \text{if } \hat{y}.\text{has\_issue} == y.\text{has\_issue}, \\ 0.0, & \textit{otherwise}, \end{cases} \tag{11}$$

where "has_issue" is the JSON keys for the QE subtask and $\hat{y}$.has_issue is the model output for the QE subtask.

IL subtask Reward $R_{\mathbf{IL}}$: The IL subtask involves generating code snippets, so we use CodeBLEU [52] to evaluate the similarity between the generated issue locations and the ground truth, in order to provide a smoother and more continuous reward signal, which is defined as:

$$R_{\text{IL}} = \frac{1}{M} \sum_{i=1}^{M} \text{CodeBLEU}(\hat{y}.\text{issues}[i].\text{issue\_location}, y.\text{issues}[i].\text{issue\_location}), \tag{12}$$

where M is the number of issues in the sample, and $\hat{y}$.issues$[i]$.issue_location is the generated issue location for the i-th issue. We average the CodeBLEU scores across all issues to obtain the final IL content reward.

IC subtask Reward $R_{\mathbf{IC}}$: The IC subtask is a multi-class classification task, so we define its content reward based on accuracy as:

$$R_{\text{IC}} = \frac{1}{M} \sum_{i=1}^{M} \mathbb{I}\left[\hat{y}.\text{issues}[i].\text{issue\_type} = y.\text{issues}[i].\text{issue\_type}\right], \tag{13}$$

where $\mathbb{I}[\cdot]$ is the indicator function that returns 1 if the condition is true and 0 otherwise and $\hat{y}$.issues$[i]$.issue_type is the generated issue type for the i-th issue. We average the accuracy across all issues to obtain the final IC content reward.

Furthermore, when multiple reference comments exist, we sort them by their locations during training, enabling the model to implicitly learn a top-to-bottom spatial prior without relying on order alignment mechanisms like Hungarian matching.

CR subtask Reward $R_{\mathbf{CR}}$: The CR subtask involves generating refined code, so we also use CodeBLEU [52] to evaluate the similarity between the generated refined code and the ground truth, which is defined as:

$$R_{\text{CR}} = \text{CodeBLEU}(\hat{y}.\text{refined\_code}, y.\text{refined\_code}), \tag{14}$$

where $\hat{y}$.refined_code is the generated refined code.

As shown above, we exclude explicit rewards for Issue Description Generation (IDG) and Fix Suggestion Generation (FSG) to prevent reward hacking [49]. Issue location and type are objective logical anchors that provide unambiguous gradient signals, whereas descriptions and fix suggestions are open-ended natural language where text similarity or proxy rewards would cause the model to overfit to rigid templates and degrade its flexible generation capabilities.

3.4.3 Curriculum-Inspired Multi-Stage Reward Fusion. The E2E-ACR paradigm comprises multiple ordered and interdependent subtasks, reflecting real-world code review workflows. Directly optimizing all subtask rewards jointly from scratch often leads to unstable training and suboptimal performance due to reward heterogeneity and the complexity of multi-reward aggregation. To mitigate this, we draw inspiration from curriculum learning [46, 71], and adopt a curriculum-inspired multi-stage reward fusion strategy. We split GRPO training into three stages of increasing difficulty and progressively expand the composition of R_c:

Stage 1: Issue Identification. The first stage only requires the model to judge whether the input code contains issues. At this stage, we retain only R_{QE}, so that policy updates are primarily

driven by the objective of issue identification. This stage aims to quickly establish a stable decision boundary and basic discriminative capability for quality defects, providing a reliable foundation for subsequent learning that requires finer-grained explanation and refinement.

Stage 2: Issue Clarification. Building on issue identification, the second stage further requires the model to clarify issue details and produce structured review outcomes. Concretely, in addition to R_{QE}, we introduce R_{IL} and R_{IC}. Without undermining the identification capability, this stage gradually extends the learning objective from a binary judgment to an explainable review conclusion, encouraging the model to generate issue localization, classification, description. This yields an information-complete intermediate reasoning chain for the final refinement.

Stage 3: Issue Resolution. After the first two stages, we further introduce R_{CR} to equip the model with issue-resolution capability, encouraging it to output actionable fix suggestions and perform code refinement. This final stage fully aligns the training objective with the full E2E-ACR paradigm and enforces a closed-loop constraint on the final output under a strong capability prior, enabling the model to learn a consistent reasoning path.

With the fusion strategy, subtasks can collaborate effectively under clearer stage-wise objectives: the model first learns "whether it can find issues," reducing noise for later stages; then learns "how to explain issues clearly and completely," forming structured and explainable review intermediates; and finally learns "whether it can fix issues correctly," completing the end-to-end loop and ultimately improving the final performance.

4 Experimental Setup

4.1 Research Questions

To comprehensively evaluate the effectiveness of E4R-Reviewer, we conduct a series of experiments to answer the following research questions (RQs):

- **RQ1:** How does E4R-Reviewer perform on each code review subtask compared with state-of-the-art baselines?
- **RQ2:** What is the quantitative and qualitative impact of the proposed logical reasoning mechanism on overall performance?
- **RQ3:** How well does E4R-Reviewer generalize to code review across multiple programming languages?
- **RQ4:** Do the outputs produced by E4R-Reviewer exhibit strong explainability and human-aligned reasoning?

Although we decompose the RCG task into four subtasks (IL, IC, IDG, and FSG) during training to obtain structured feedback, we evaluate RCG as a single task to better reflect the model's ability to generate complete review comments. When multiple comments are generated, we merge them for holistic assessment.

4.2 Datasets

SFT and GRPO training dataset. We use the Carllm training set [77], comprising 19,456 annotated code review instances, and further augment it as described in Sec. 3.2.1 to include issue type, refined code, and reasoning process annotations aligned with our task formulation.

Evaluation datasets. For QE task, we use the Carllm test set [77], which contains 2,000 samples with balanced positive and negative labels. For RCG and CR tasks, we adopt the CodeReviewer test set [38], which provides widely-accepted evaluation protocols enabling fair comparison with existing work. From this dataset, We randomly sample 400 instances per language across five programming languages (Python, C/C++, Java, JavaScript, Go), yielding 2,000 evaluation instances in total. Each includes ground-truth review comments and refined code annotations.

4.3 Evaluation Criteria

We adopt both quantitative and qualitative metrics to comprehensively evaluate model performance.

Quality Estimation. For the QE task, we report *Accuracy*, *Precision*, *Recall*, and *F1-score*, which are standard metrics for classification.

Review Comments Generation. For the RCG task, multiple studies have shown that review comments in the CodeReviewer dataset suffer from quality issues [35, 57, 79], rendering surface-level text-overlap metrics (e.g., BLEU, ROUGE) unreliable for assessing actual usefulness. Recent work indicates that LLM-based evaluation aligns well with human judgment [36, 80]; therefore, we adopt the LLM evaluation protocol proposed by Zhang et al. [79] to assess comment quality along five criteria:

- C1. Readability: Clear, easily understandable language.
- C2. Relevance: Directly related to the specific code.
- C3. Brevity: Clearly explain the issues identified, and at the same time, be concise and not lengthy.
- C4. Sufficiency: Cover all issues as much as possible, and point out the exact location of issues for comprehensive review.
- C5. Operability: Practical advice for addressing identified issues.

A 5-point Likert scale is adopted to balance granularity and reliability, avoiding the coarseness of 3-point scales and the instability of 10-point scales [30]. Additionally, due to the linguistic diversity and high variability of LLM outputs [4], we follow Li et al. [38] and conduct a manual evaluation in which annotators rate each comment on two dimensions using a 5-point Likert scale:

- Relevance: Whether comments precisely identify defect locations and root causes.
- Information: Degree to which the comment provides sufficient rationale and context to make the issues and suggestions easy to understand.

Code Refinement. For the CR task, we use *BLEU* and *CodeBLEU* as automated metrics, and complement them with both LLM and human evaluation to measure the quality and correctness of the generated code. Both LLM and human evaluation are formulated as binary decisions: whether the generated code resolves the quality issues in the original code and is an improvement over it. We then report the corresponding accuracies, denoted as LLM_Acc and H_Acc.

For LLM evaluation, we use GPT-5 (gpt-5-2025-08-07) as an automatic judge to ensure consistent assessments and avoid overlap with any evaluated baseline. For human evaluation, two software engineering master's students—each with at least four years of professional development and code review experience—serve as annotators, working independently on all sampled instances. We randomly sample 10% of the test instances (200 in total) for annotation. We assess inter-annotator agreement using Cohen's Kappa (κ) [42] for binary ratings and Weighted Kappa (κ_w) [76] with quadratic weights for ordinal Likert ratings, and also report the agreement between the LLM evaluator and each human annotator. The final score per instance is the average of the two annotators' ratings.

4.4 Baselines

We compare E4R-Reviewer with a comprehensive set of baselines, grouped into two categories: (i) ACR models and (ii) general-purpose LLMs.

ACR models. For DL-based ACR models, we include CodeBERT [12], CodeT5 [72], T5-Reviewer [68], CodeReviewer [38] and DISCOREV [5]. For LLM-based ACR models, we evaluate LLaMA-Reviewer [41] and the Carllm series [77].

DISCOREV is included as a baseline for the RCG and CR tasks, as it improves these tasks through static cross-task distillation. We exclude LAURA [79] because it relies on PR-level context (e.g., conversation history and reviewer profiles), which is unavailable in our setting. As a complementary comparison, we additionally evaluate advanced few-shot LLMs below, which partially capture exemplar-based enhancement without such additional information.

General LLMs. We consider both open-source and proprietary models. Specifically, the open-source models include: (1) the **LLaMA** series (LLaMA-13B [66], LLaMA-2-13B [67], and CodeLLaMA-13B [53]), (2) the **Qwen** series (Qwen-2.5-Coder-7B/32B-Instruct [27]), and (3) the **DeepSeek** series (DeepSeek-Coder-6.7B/33B-Instruct [17] and DeepSeek-V3 [40]). The proprietary models include GPT-4o [28], Gemini-2.5-Pro [9], and Claude-3.7-Sonnet [1].

For ACR baselines, we strictly follow the original implementations and hyperparameter settings to ensure a fair comparison. For general LLM baselines, we adopt the same task prompts as E4R-Reviewer and evaluate them in a 2-shot setting, using one positive (issue-present) and one negative (issue-absent) demonstration.

4.5 Implementation Details

Backbone. We use Qwen-2.5-Coder-7B-Instruct as the backbone model. The 7B scale offers a practical balance between capacity and efficiency and mitigates overfitting on relatively small training sets.

Parameter-efficient fine-tuning. We adopt QLoRA [11] for efficient adaptation: the backbone weights are quantized to 4-bit precision during loading and the adapter rank is set to r=8 with a scaling factor of 16.

Training setup. Due to resource constraints, we cap the maximum input length at 2,048 tokens. Training proceeds in two stages—SFT followed by GRPO-based RL—with identical batching: per-device batch size of 8 and gradient accumulation over 4 steps. We use 8-bit AdamW (β_1=0.9, β_2=0.99) with weight decay 1×10^{-2}. For SFT, we train for 1 epoch with learning rate 1×10^{-4}; for GRPO, we train for 4 epochs with learning rate 1×10^{-5} and sample 4 candidate completions per prompt for policy optimization. For the multi-stage reward fusion, we allocate 1, 1, and 2 epochs to stages 1-3, respectively. The content-reward weights $(\lambda_{\text{QE}}, \lambda_{\text{IL}}, \lambda_{\text{IC}}, \lambda_{\text{CR}})$ are set to $[1.0, 0.0, 0.0, 0.0]$, $[0.6, 0.2, 0.2, 0.0]$, and $[0.2, 0.1, 0.1, 0.6]$ for stages 1-3. We set the weights of the content reward λ_c and the format λ_f reward to 0.7 and 0.3, respectively.

Hardware and frameworks. All experiments are run on a single NVIDIA H200 GPU (140GB). We use Unsloth [70] to implement SFT and GRPO training and vLLM [32] for fast inference.

5 Experimental Results

5.1 RQ1: Performance on Code Review subtasks

5.1.1 Quality Estimation Task. As shown in Table 1, E4R-Reviewer outperforms all baselines on the QE task across all metrics. Compared to the strongest baseline, CodeCarllm-13B, it achieves absolute gains of +1.85% (Accuracy), +2.17% (Precision), +0.60% (Recall), and +1.45% (F1). Notably, E4R-Reviewer uses only ~60% of CodeCarllm-13B's parameters, demonstrating superior parameter efficiency. Against the size-comparable MagicCarllm-7B, it improves Accuracy by +4.30% and F1 by +9.26%.

From a methodological perspective, DL-based ACR models consistently underperform LLM-based approaches in F1, suggesting that generic code representations or seq2seq frameworks struggle to capture the fine-grained signals needed to judge whether a change is warranted. In contrast, LLM-based ACR models achieve higher recall and overall QE performance, enabling more proactive issue detection. General-purpose LLMs perform poorly on QE, indicating that this task relies

less on surface-level language understanding and more on systematic knowledge of engineering conventions, code smells, maintainability, and defect patterns. Moreover, standard LLMs are ill-suited for binary classification: some exhibit extreme "high-recall, low-precision" behavior—for instance, Qwen-2.5-Coder-32B-Instruct attains 99.20% Recall by labeling nearly all samples as problematic, but its Precision collapses to 50.15%.

In contrast, E4R-Reviewer strikes a balanced trade-off: it maintains high Recall for broad coverage while significantly improving Precision to suppress false alarms from stylistic or non-critical differences, yielding both higher F1 and more reliable performance.

Table 1. Performance Comparison on Quality Estimation Task

Model Type	**Model**	Acc (%)	P (%)	R (%)	F1 (%)
ACR Models	CodeBERT	66.95	70.49	58.30	63.82
	CodeT5	66.55	68.28	61.80	64.88
	T5-Reviewer	66.20	67.85	61.15	64.71
	CodeReviewer	67.80	70.84	60.50	65.26
	LLaMA-Reviewer	63.80	60.66	78.50	68.44
	MagicCarllm-7B	69.30	75.00	57.90	65.35
	Carllm-13B	67.20	62.68	85.00	72.15
	Carllm2-13B	68.95	68.74	69.50	69.12
	CodeCarllm-13B	71.75	69.68	77.00	73.16
General LLMs	LLaMA-13B	48.35	48.55	55.20	51.66
	LLaMA-2-13B	50.75	51.13	33.80	40.70
	CodeLLaMA-13B	48.35	48.55	55.20	51.66
	Qwen-2.5-Coder-7B-Instruct	50.20	50.43	93.30	65.47
	Qwen-2.5-Coder-32B-Instruct	50.30	50.15	99.20	66.62
	DeepSeek-Coder-6.7B-Instruct	46.40	49.14	85.60	62.44
	DeepSeek-Coder-33B-Instruct	49.30	50.19	79.60	61.56
	DeepSeek-V3	61.90	66.57	47.80	55.65
	GPT-4o	56.15	53.71	89.10	67.02
	Gemini-2.5-Pro	61.95	60.79	67.30	63.88
	Claude-3.7-Sonnet	54.10	53.61	60.90	57.02
Ours	**E4R-Reviewer**	**73.60**	**71.85**	**77.60**	**74.61**

5.1.2 Review Comments Generation Task. After completing the QE task, we first filtered out models that performed poorly or were unable to consistently produce high-quality outputs. Only models that passed this initial screening were included in the subsequent evaluation stages. We then conducted both LLM and human evaluation, with the results summarized in Table 2.

Overall, E4R-Reviewer achieved the best performance across all five LLM evaluation dimensions. Its scores for C1–C5 remain consistently around 4.0, demonstrating strong effectiveness in generating review comments. DISCOREV, which also explores cross-task knowledge transfer via distillation, scores slightly better than other ACR models but substantially lower than E4R-Reviewer, confirming that our end-to-end reasoning-guided approach yields more informative and actionable review comments. In the C3. Brevity, the performance gap between E4R-Reviewer

and other baselines is notably smaller. Since brevity focuses primarily on concise expression, the differences across models tend to be less amplified than in content-oriented dimensions, making this trend expected. Although DL-based ACR models (e.g., CodeReviewer) typically generate shorter unstructured texts, they score relatively low on brevity. One possible explanation is that the brevity metric first requires precise issue localization before conciseness is rewarded. During batch evaluation, all General LLMs output structured comments with explicit field annotations indicating issue locations, putting ACR models at a disadvantage in this dimension.

Human evaluation results exhibit a trend consistent with LLM evaluation. E4R-Reviewer achieves the best performance, indicating that its generated review comments better align with human expectations in terms of relevance and informativeness, which improving the effectiveness and explainability. The inter-annotator agreement reaches κ_w=0.83 (Relevance) and 0.88 (Information), and the LLM–human agreement reaches an avg. κ_w of 0.84.

Table 2. Performance Comparison on Review Comments Generation Task (C* denotes the five LLM evaluation criteria).

Model	**LLM Eval.**					**Human Eval.**	
	C1	C2	C3	C4	C5	Relevance	Information
T5-Reviewer	2.59	2.13	2.69	2.35	2.57	–	–
CodeReviewer	2.54	2.70	2.74	2.40	2.38	3.02	2.54
LLaMA-Reviewer	2.85	2.61	2.60	2.57	2.51	–	–
DISCOREV	2.52	2.67	2.66	2.45	2.43	3.05	2.66
Qwen-2.5-Coder-7B-Instruct	3.23	3.44	3.24	3.33	3.31	–	–
Qwen-2.5-Coder-32B-Instruct	3.83	4.04	3.88	3.93	3.90	3.68	3.77
DeepSeek-Coder-6.7B-Instruct	3.11	3.22	3.07	3.12	3.14	–	–
DeepSeek-Coder-33B-Instruct	3.48	3.58	3.45	3.46	3.51	3.21	3.66
GPT-4o	3.69	3.84	3.41	3.69	3.45	3.43	3.64
Gemini-2.5-Pro	3.29	3.41	3.33	3.35	3.35	–	–
E4R-Reviewer	**4.03**	**4.19**	**3.97**	**4.14**	**4.13**	**3.78**	**3.89**

5.1.3 Code Refinement Task. In the CR task, traditional ACR models require both the code diff and a human-written review comment as input. This setting is fundamentally different from our task paradigm, which exclusively takes the raw code diff as input. Due to this mismatch, we focus our evaluation on general LLMs, and the results are presented in Table 3.

According to the automatic evaluation metrics (BLEU and CodeBLEU), E4R-Reviewer significantly outperforms all baseline models. It achieves 65.59 BLEU and 58.75 CodeBLEU, outperforming the second-best model (48.51 BLEU, 47.78 CodeBLEU) by 17.08 and 10.97 points, respectively. These improvements demonstrate that E4R-Reviewer generates refined code that is not only more aligned with the reference implementation but also shows greater semantic and structural consistency. Although DISCOREV also considers inter-task dependencies, it scores much lower (32.35 BLEU and 35.60 CodeBLEU) when receiving upstream RCG outputs instead of ground-truth comments, confirming that our end-to-end reasoning chain models inter-task dependencies more effectively than static pairwise distillation.

A consistent pattern emerges in both LLM and human evaluations. E4R-Reviewer attains the highest LLM_Acc (45.30) and H_Acc (41.25), clearly outperforming strong general models such as

GPT-4o and Gemini-2.5-Pro. The inter-annotator agreement reaches κ=0.91, and the LLM–human agreement reaches an avg. κ of 0.87. These findings highlight the effectiveness of domain-specialized modeling through reinforcement learning, particularly in diff-driven code refinement scenarios.

Table 3. Performance Comparison on Code Refinement Task

Model	Automatic Eval.		LLM Eval.	Human Eval.
	BLEU	CodeBLEU	LLM_Acc (%)	H_Acc (%)
DISCOREV	32.35	35.60	21.30	23.75
Qwen-2.5-Coder-7B-Instruct	26.80	36.19	13.50	–
Qwen-2.5-Coder-32B-Instruct	43.94	44.79	34.90	30.25
DeepSeek-Coder-6.7B-Instruct	39.04	40.56	18.60	–
DeepSeek-Coder-33B-Instruct	31.45	39.72	19.75	23.50
GPT-4o	42.90	42.93	37.80	38.75
Gemini-2.5-Pro	48.51	47.78	35.75	–
E4R-Reviewer	**65.59**	**58.75**	**45.30**	**41.25**

5.1.4 Statistical Significance. We conduct 5 independent trials with different random seeds (or adjusting temperature settings for proprietary models) and perform Wilcoxon rank-sum tests against the SOTA baseline (i.e., CodeCarllm-13B for QE and Gemini-2.5-Pro for CR) on the QE and CR tasks. As shown in Table 4, all improvements are statistically significant: QE Accuracy ($p < 0.01$) and F1 ($p < 0.05$), CR BLEU and CodeBLEU ($p < 0.01$), confirming that our gains are robust across multiple trials.

Table 4. Statistical Significance of E4R-Reviewer vs. SOTA Baseline on QE and CR Tasks.

Model	QE Task		CR Task	
	Acc (%)	F1 (%)	BLEU	CodeBLEU
SOTA Baseline	72.04 ± 0.47	72.81 ± 0.60	48.28 ± 0.96	47.80 ± 1.07
E4R-Reviewer	73.82 ± 0.44 (**)	74.80 ± 0.65 (*)	66.19 ± 1.63 (**)	58.27 ± 1.46 (**)

Note: Values are reported as mean ± std. Significance levels are denoted by * ($p < 0.05$) and ** ($p < 0.01$).

5.2 RQ2: Ablation Study of E4R-Reviewer

To analyze the contribution of each component in E4R-Reviewer, we conduct an ablation study on the QE and CR tasks, which represent the beginning and end of the code review pipeline and admit reliable automatic metrics. Results are reported in Table 5.

Starting from the backbone (V1), applying SFT (V2) consistently improves both tasks, suggesting that limited supervision already enhances basic task alignment. Building upon V2, adding task-specific reward functions with GRPO-based RL (V3, V4) yields substantial gains on the targeted tasks: V3 improves the Acc and F1, while V4 improves the BLEU and CodeBLEU, confirming the effectiveness of GRPO-based RL, which consistently further improves task performance over SFT under single-task settings.

When all task rewards are unified for joint training (V5), CR metrics further improve over V4, indicating benefits from cross-task knowledge sharing. However, QE shows a precision–recall shift (higher recall, lower precision) with F1 similar to V3, implying a tendency to over-predict issues. We attribute this to reward heterogeneity: QE uses a strict binary reward, whereas downstream tasks provide denser, softer preference rewards. As a result, during joint training, the model receives richer and more frequent gradient signals from later tasks, causing QE—despite being the first step of the pipeline—to receive insufficient optimization pressure.

Finally, we apply curriculum-inspired multi-stage reward fusion (V6). Following the real-world code review workflow, we divide multi-reward fusion into three stages to mitigate the issues caused by the naive reward combination in V5. As a result, V6 outperforms V5 on both QE and CR, validating the effectiveness of multi-stage reward fusion. Moreover, compared with the single-reward variants (V3 and V4), V6 further improves the corresponding tasks, highlighting the benefits of multi-task modeling.

Table 5. Ablation Study on E4R-Reviewer for QE and CR Tasks (V* denotes different ablation model variants).

Version	Model	QE Task				CR Task	
		Acc (%)	P (%)	R (%)	F1 (%)	BLEU	CodeBLEU
V1	Qwen-2.5-Coder-7B-Ins.	50.20	50.43	93.30	65.47	26.80	36.19
V2	V1 + SFT	54.50	52.38	99.20	68.56	38.94	40.76
V3	V2 + GRPO (QE Reward)	73.10	71.67	76.40	73.96	40.14	41.44
V4	V2 + GRPO (CR Reward)	55.95	53.18	99.40	69.29	61.36	55.30
V5	V2 + GRPO (All Reward)	68.20	63.30	86.60	73.14	63.23	56.97
V6 (Ours)	V5 + CL (**E4R-Reviewer**)	**73.60**	**71.85**	**77.60**	**74.61**	**65.59**	**58.75**

To further illustrate the stability of RL training and the effectiveness of curriculum learning, we present the reward curves for format, QE, and CR tasks in Figure 5 for Curriculum Learning (CL) and Joint Training (JT). For R_f, both CL and JT start from a high initial reward after SFT and converge smoothly, indicating that SFT already equips the model with strong format-following capability, and the fixed weight in the total reward ensures stability. For R_{QE}, CL focuses solely on QE in Stage 1, where the reward rises rapidly; it continues to improve in Stage 2 and Stage 3. In contrast, JT mixes signals from multiple tasks from the start, resulting in suboptimal QE learning and a lower final reward than CL. For R_{CR}, since CL does not include the R_{CR} in Stages 1 and 2 (which focus on issue identification and clarification), the model only receives R_{CR} during Stage 3. Despite training on R_{CR} for only two epochs, the model learns CR more efficiently after mastering preceding tasks: the ascending slope is steeper and the final level is higher than JT. This demonstrates that curriculum learning enables effective knowledge transfer—mastery of earlier tasks provides a stronger foundation for subsequent, more complex tasks.

5.3 RQ3: Generalization Across Programming Languages

In this section, we analyze cross-language generalization across programming languages. Table 6 reports QE and CR results for five training-covered languages (Python, C/C++, Java, JavaScript, and Go), measured by F1 and CodeBLEU (reported as F1/CodeBLEU).

Overall, E4R-Reviewer achieves the best performance across all languages, clearly outperforming GPT-4o and the Qwen-2.5-Coder series. For instance, on C/C++ and Go it attains F1 scores of

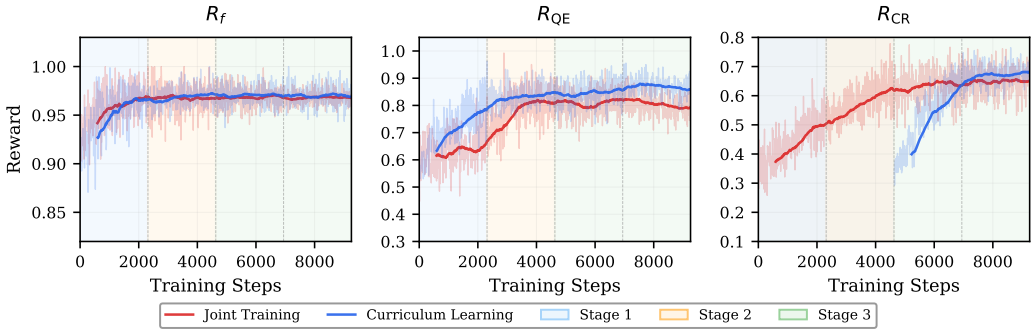

Fig. 5. Reward curves for R_f, R_{QE}, and R_{CR} under Curriculum Learning and Joint Training.

85.12 and 81.79 with CodeBLEU of 56.94 and 61.14, respectively, indicating robust generalization across diverse syntax and engineering conventions.

Across languages, E4R-Reviewer shows a more balanced performance in CodeBLEU, suggesting that its generated edits preserve relatively high code consistency and readability across languages. In contrast, its F1-scores on Python and JavaScript are slightly lower than those on C/C++, Java, and Go. Further analysis suggests that this gap mainly stems from lower Precision: the model is more inclined to predict "has_issues", leading to more false positives. We attribute this phenomenon to both the training data distribution and language characteristics. First, Python/JavaScript samples in the training set more frequently involve issues related to naming, formatting, weak typing, and asynchronous logic, which encourages the model to apply stricter judgments and thus increases false positives. Second, Python/JavaScript syntax and coding styles are more flexible, making the "needs modification" decision more subjective and prone to annotation noise, which further reduces Precision.

Table 6. Per-language Performance on QE (F1) and CR (CodeBLEU) Tasks [F1/CodeBLEU]

Model	Python	C/C++	Java	JavaScript	Go
Qwen-2.5-Coder-7B-Ins.	58.57/36.75	70.15/36.58	72.23/38.19	52.84/37.09	76.50/32.35
Qwen-2.5-Coder-32B-Ins.	58.20/41.61	73.90/43.88	74.04/44.98	53.24/43.73	78.06/49.86
GPT-4o	60.81/41.45	74.33/42.81	74.10/43.75	53.91/41.56	75.81/45.18
E4R-Reviewer	**63.75/60.89**	**85.12/56.94**	**79.13/61.08**	**60.00/53.60**	**81.79/61.14**

5.4 RQ4: Qualitative Analysis and Case Studies

5.4.1 Effectiveness and Explainability. To qualitatively demonstrate the effectiveness and explainability of E4R-Reviewer, we present a representative case study in Fig. 6. The code change introduces a new condition to filter titles that do not start with either a backslash (\) or a forward slash (/). However, the developer accidentally misspelled the Python string method `startswith` as `strartswith`, which would raise an `AttributeError` at runtime.

As shown in the figure, baseline models such as T5-Reviewer, CodeReviewer, and LLaMA-Reviewer either provide vague feedback (e.g., "I think this is a bug") or fail to identify the root cause. Notably, Qwen-2.5-7B-Instruct incorrectly treats `strartswith` as a valid method and suggests keeping the erroneous logic, potentially misleading developers.

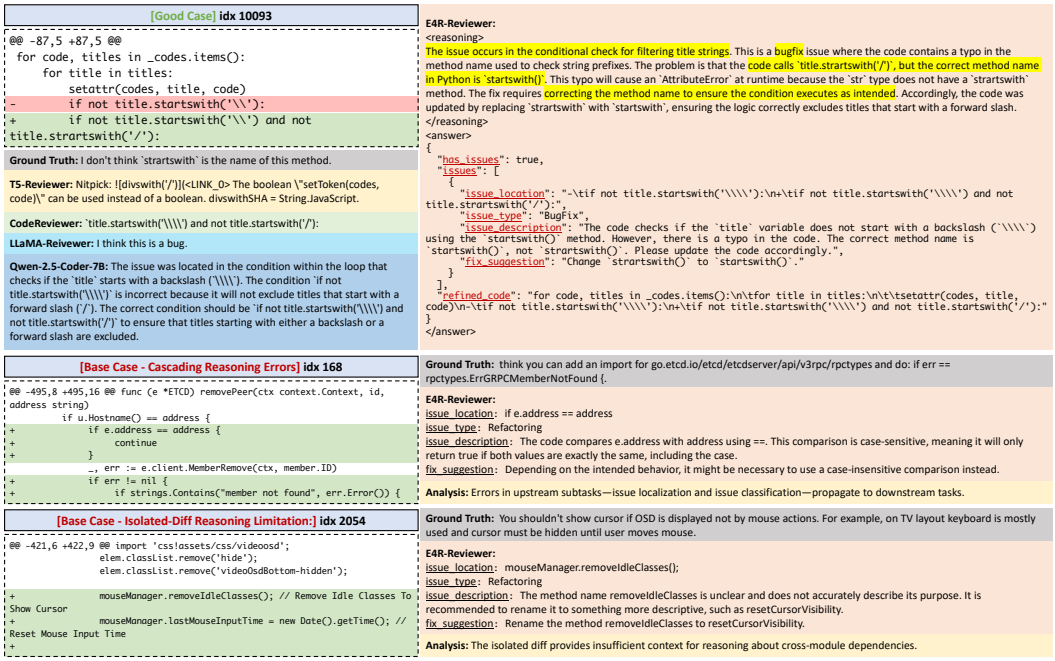

Fig. 6. Case Studies of E4R-Reviewer: A Good Case and Two Typical Error Cases

In contrast, E4R-Reviewer precisely localizes the issue, identifies it as a *BugFix*-type problem, and clearly explains that "the correct method name is `startswith()`, not `strartswith()`." It further provides a concrete fix suggestion: "Change `strartswith()` to `startswith()`." This reasoning reflects our model's end-to-end multi-task understanding—spanning issue localization, type classification, semantic description, and actionable refinement—while maintaining high interpretability. The case illustrates how E4R-Reviewer not only detects subtle syntactic errors but also communicates them in a developer-friendly manner, thereby reducing cognitive load and building trust.

5.4.2 Error Analysis and Limitations. To gain deeper insight into the limitations of E4R-Reviewer, we perform a thorough error analysis of review comments that received low human evaluation scores. Following the issue classification taxonomy defined in the IC subtask, we find that failures remain predominantly concentrated on functional modifications—BugFix and Refactoring—which require deeper semantic understanding of code intent and control flow. We further identify two primary error patterns, as illustrated in Fig. 6:

Cascading Reasoning Errors: Since E4R-Reviewer models code review as an end-to-end reasoning chain, errors at earlier stages can propagate downstream. For instance, the model mislocalizes the issue (pointing to an address-check line instead of the error-handling line), misclassifies a BugFix-type issue as Refactoring, and consequently generates an incorrect fix suggestion based on string comparison rather than the proper error-type check. This cascading effect compounds the error across multiple subtasks. To mitigate this, incorporating stage-level self-verification mechanisms or introducing human feedback at critical decision points could help the model self-correct before errors propagate.

Isolated-Diff Reasoning Limitation: The model struggles with issues that require understanding cross-module dependencies and broader codebase context, which cannot be inferred from the isolated code diff alone. For example, when reviewing a UI component change involving cursor visibility, the model fails to recognize that the correct behavior depends on the interaction between `mouseManager` and `layoutManager` across separate modules. Without access to the broader module interactions, E4R-Reviewer cannot capture such cross-module logic. To address this, integrating repository-level context through retrieval-augmented generation or leveraging static analysis tools could enable the model to reason beyond the local diff.

6 Related Work

6.1 Automated Code Review

Given the importance of code review for software quality and the cost of manual review, researchers have developed ACR techniques. Early work focused on discriminative tasks such as reviewer recommendation [31, 44, 64], comment classification [39, 47], and quality estimation [22, 23, 59].

With advances in natural language processing, research shifted to generative tasks. Early approaches used Information Retrieval (IR), e.g., DeepCodeReviewer [20] and CommentFinder [24], which retrieved comments from historical pools, while CORE [62] enhanced semantic matching via multi-level embeddings. However, IR methods are limited by candidate pool coverage and generalize poorly to novel changes.

DL methods later became dominant. T5-based models showed strong performance [68, 69], and CodeReviewer [38] extended CodeT5 [72] via multi-task pre-training to support end-to-end quality estimation, comment generation, and code refinement. DISCOREV [5] exploited cross-task couplings through knowledge distillation. Recently, LLM-based approaches emerged: LLaMA-Reviewer [41] adapted LLaMA [66] via Parameter-Efficient Fine-Tuning (PEFT) [25], and Carllm [77] introduced structured comments for better explainability.

Despite progress, existing ACR methods fail to fully model the logical order and internal dependencies among subtasks and often lack explainability. E4R-Reviewer bridges these gaps with a unified, end-to-end, reasoning-guided alignment framework, improving effectiveness by leveraging cross-task information and explicitly producing explainable outputs.

6.2 Reinforcement Learning for Large Reasoning Models

RL [65] achieved early success in rule-based games, mastering Go and chess (e.g., AlphaGo [60], AlphaZero [61]) via self-play. With the rise of LLMs, RL shifted toward human alignment through methods like RLHF [8] and DPO [51].

Recently, RL with Verifiable Rewards (RLVR) [33] has emerged as a paradigm for training reasoning LLMs [78], incentivizing extended reasoning for complex tasks (e.g., OpenAI o1 [29], DeepSeek-R1 [16]). In SE, RLVR has been applied to program repair [14, 26] and code generation [73].

However, in the ACR domain, despite a few early explorations of RL [56], no prior work has applied RL to LLM-based ACR with explicit reasoning guidance. E4R-Reviewer builds on an LLM backbone and adopts GRPO-based RL with a curriculum-guided, multi-stage reward fusion, unlocking the potential of RL for LLM-based ACR models.

7 Threats to Validity

Construct Validity. Automated metrics like BLEU and CodeBLEU capture only surface-level similarity and fail to reflect the reasoning quality and explainability of human-like review comments.

To address this, we complement them with both LLM and human evaluations of generated comments and refined code, better aligning assessment with human preferences and improving evaluation comprehensiveness.

Internal Validity. To reduce LLM stochasticity, we fix the random seed and use temperature 0 for primary experiments. Although pretraining data leakage cannot be fully excluded, we mitigate it via strict dataset partitioning, and rigorously verify the train/test split exhibits no data leakage or distribution bias. For fair comparison, we follow the original implementations and hyperparameter settings of all baselines.

External Validity. Due to resource constraints, experiments are conducted on a public open-source dataset spanning five mainstream languages and 7B-scale models, which may limit generalization to larger models and broader settings. Future work will extend evaluation to more languages and model scales to further assess robustness.

8 Conclusion

We propose E4R-Reviewer, enhancing the effectiveness and explainability of ACR via end-to-end reasoning-guided alignment. Unlike prior paradigms modeling subtasks in isolation, E4R-Reviewer formulates code review as an end-to-end reasoning process aligned with real-world review workflows, and leverages GRPO-based RL alignment to turn reasoning steps into optimizable intermediate objectives. We further design a curriculum-inspired multi-stage reward fusion mechanism to better exploit the logical order and dependencies across subtasks, improving overall effectiveness. Meanwhile, E4R-Reviewer strengthens explainability and usability by producing intermediate reasoning traces and structured review outputs. Although implemented with a specific LLM backbone in our experiments, E4R-Reviewer is inherently model-agnostic, readily adaptable to other LLMs, and may reach higher performance ceilings with larger model and data scales.

9 Data Availability

All the source code and data are publicly available at https://zenodo.org/records/18417185.

References

[1] Anthropic. 2025. Claude 3.7 Sonnet. https://www.anthropic.com/news/claude-3-7-sonnet

[2] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 712–721.

[3] Vipin Balachandran. 2013. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 931–940.

[4] Yejin Bang. 2023. A Multitask, Multilingual, Multimodal Evaluation of ChatGPT on Reasoning, Hallucination, and Interactivity. *arXiv preprint arXiv:2302.04023* (2023).

[5] Oussama Ben Sghaier and Houari Sahraoui. 2024. Improving the learning of code review successive tasks with cross-task knowledge distillation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1086–1106.

[6] Amiangshu Bosu, Michaela Greiler, and Christian Bird. 2015. Characteristics of useful code reviews: An empirical study at microsoft. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 146–156.

[7] Junkai Chen, Zhenhao Li, Qiheng Mao, Xing Hu, Kui Liu, and Xin Xia. 2025. Understanding Practitioners' Expectations on Clear Code Review Comments. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1257–1279.

[8] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. *Advances in neural information processing systems* 30 (2017).

[9] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261* (2025).

[10] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. 2015. Code reviews do not find bugs. how the current code review best practice slows us down. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 27–28.

[11] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems* 36 (2023), 10088–10115.
[12] Z Feng. 2020. Codebert: A pre-trained model for program-ming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
[13] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*. 935–947.
[14] Marcos Fuster-Pena, David de Fitero-Dominguez, Antonio Garcia-Cabot, and Eva Garcia-Lopez. 2025. Repaca: Leveraging reasoning large language models for static automated patch correctness assessment. *arXiv preprint arXiv:2507.22580* (2025).
[15] Ian X Gauthier, Maxime Lamothe, Gunter Mussbacher, and Shane McIntosh. 2021. Is historical data an appropriate benchmark for reviewer recommendation systems?: A case study of the gerrit community. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 30–41.
[16] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).
[17] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).
[18] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2024. Exploring the potential of chatgpt in automated code refinement: An empirical study. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.
[19] Qi Guo, Xiaofei Xie, Shangqing Liu, Ming Hu, Xiaohong Li, and Lei Bu. 2025. Intention is all you need: Refining your code from your intention. *arXiv preprint arXiv:2502.08172* (2025).
[20] Anshul Gupta and Neel Sundaresan. 2018. Intelligent code reviews using deep learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18) Deep Learning Day*.
[21] Ahmad Haji Mohammadkhani. 2023. Explainable AI for Software Engineering: A Systematic Review and an Empirical Study. (2023).
[22] Vincent J Hellendoorn, Jason Tsay, Manisha Mukherjee, and Martin Hirzel. 2021. Towards automating code review at scale. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1479–1482.
[23] Haytham Hijazi, Joao Duraes, Ricardo Couceiro, Joao Castelhano, Raul Barbosa, Júlio Medeiros, Miguel Castelo-Branco, Paulo De Carvalho, and Henrique Madeira. 2022. Quality evaluation of modern code reviews through intelligent biometric program comprehension. *IEEE Transactions on Software Engineering* 49, 2 (2022), 626–645.
[24] Yang Hong, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, and Aldeida Aleti. 2022. 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*. 507–519.
[25] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *ICLR* 1, 2 (2022), 3.
[26] Haichuan Hu, Xiaochen Xie, and Quanjun Zhang. 2025. Repair-r1: Better test before repair. *arXiv preprint arXiv:2507.22853* (2025).
[27] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).
[28] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).
[29] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720* (2024).
[30] Ankur Joshi, Saket Kale, Satish Chandel, and D Kumar Pal. 2015. Likert scale: Explored and explained. *British journal of applied science & technology* 7, 4 (2015), 396–403.
[31] Dezhen Kong, Qiuyuan Chen, Lingfeng Bao, Chenxing Sun, Xin Xia, and Shanping Li. 2022. Recommending code reviewers for proprietary software projects: A large scale study. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 630–640.
[32] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*. 611–626.
[33] Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, et al. 2024. Tulu 3: Pushing frontiers in open language model post-training.

arXiv preprint arXiv:2411.15124 (2024).

[34] Lingwei Li, Li Yang, Huaxi Jiang, Jun Yan, Tiejian Luo, Zihan Hua, Geng Liang, and Chun Zuo. 2022. AUGER: automatically generating review comments with pre-training models. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1009–1021.

[35] Shuochuan Li, Dong Wang, Patanamon Thongtanunam, Zan Wang, Jiuqiao Yu, and Junjie Chen. 2025. Issue-Oriented Agent-Based Framework for Automated Review Comment Generation. *arXiv preprint arXiv:2511.00517* (2025).

[36] Xuechen Li, Tianyi Zhang, Yann Dubois, Rohan Taori, Ishaan Gulrajani, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. 2023. Alpacaeval: An automatic evaluator of instruction-following models.

[37] Yingling Li, Yuhan Wu, Zi'ao Wang, Lei Huang, Junjie Wang, Jianping Li, and Minying Huang. 2025. CodeDoctor: multi-category code review comment generation. *Automated Software Engineering* 32, 1 (2025), 25.

[38] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. 2022. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1035–1047.

[39] Zhixing Li, Yue Yu, Gang Yin, Tao Wang, Qiang Fan, and Huaimin Wang. 2017. Automatic Classification of Review Comments in Pull-based Development Model.. In *SEKE*. 572–577.

[40] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[41] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 647–658.

[42] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.

[43] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2014. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th working conference on mining software repositories*. 192–201.

[44] Ehsan Mirsaeedi and Peter C Rigby. 2020. Mitigating turnover with code review recommendation: Balancing expertise, workload, and knowledge distribution. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 1183–1195.

[45] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. 2015. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE, 171–180.

[46] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E Taylor, and Peter Stone. 2020. Curriculum learning for reinforcement learning domains: A framework and survey. *Journal of Machine Learning Research* 21, 181 (2020), 1–50.

[47] Miroslaw Ochodek, Miroslaw Staron, Wilhelm Meding, and Ola Söder. 2022. Automated code review comment classification to improve modern code reviews. In *International Conference on Software Quality*. Springer, 23–40.

[48] Doriane Olewicki, Sarra Habchi, and Bram Adams. 2024. An empirical study on code review activity prediction and its impact in practice. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2238–2260.

[49] Romain Paulus, Caiming Xiong, and Richard Socher. 2017. A deep reinforced model for abstractive summarization. *arXiv preprint arXiv:1705.04304* (2017).

[50] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2024. Fine-tuning and prompt engineering for large language models-based code review automation. *Information and Software Technology* 175 (2024), 107523.

[51] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems* 36 (2023), 53728–53741.

[52] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

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

[54] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*. 181–190.

[55] Neela Sawant and Srinivasan H Sengamedu. 2023. Code compliance assessment as a learning problem. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 445–454.

[56] Oussama Ben Sghaier, Rosalia Tufano, Gabriele Bavota, and Houari Sahraoui. 2025. Leveraging Reward Models for Guiding Code Review Comment Generation. *arXiv preprint arXiv:2506.04464* (2025).
[57] Oussama Ben Sghaier, Martin Weyssow, and Houari Sahraoui. 2025. Harnessing Large Language Models for Curated Code Reviews. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 187–198.
[58] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300* (2024).
[59] Shu-Ting Shi, Ming Li, David Lo, Ferdian Thung, and Xuan Huo. 2019. Automatic code review by learning the revision of source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 4910–4917.
[60] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.
[61] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815* (2017).
[62] Jing Kai Siow, Cuiyun Gao, Lingling Fan, Sen Chen, and Yang Liu. 2020. Core: Automating review recommendation for code changes. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 284–295.
[63] Davide Spadini, Gül Çalikli, and Alberto Bacchelli. 2020. Primers or reminders? the effects of existing review comments on code review. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1171–1182.
[64] Emre Sülün. 2019. Suggesting reviewers of software artifacts using traceability graphs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1250–1252.
[65] Richard S Sutton, Andrew G Barto, et al. 1998. *Reinforcement learning: An introduction*. Vol. 1. MIT press Cambridge.
[66] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
[67] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
[68] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using pre-trained models to boost code review automation. In *Proceedings of the 44th international conference on software engineering*. 2291–2302.
[69] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. 2021. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 163–174.
[70] UnslothAI. [n. d.]. unsloth. GitHub repository. https://github.com/unslothai/unsloth
[71] Xin Wang, Yudong Chen, and Wenwu Zhu. 2021. A survey on curriculum learning. *IEEE transactions on pattern analysis and machine intelligence* 44, 9 (2021), 4555–4576.
[72] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
[73] Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. Rlcoder: Reinforcement learning for repository-level code completion. *arXiv preprint arXiv:2407.19487* (2024).
[74] Ratnadira Widyasari, Ting Zhang, Abir Bouraffa, Walid Maalej, and David Lo. 2025. Explaining explanations: An empirical study of explanations in code reviews. *ACM Transactions on Software Engineering and Methodology* 34, 6 (2025), 1–30.
[75] Bingting Wu and Xiaofang Zhang. 2022. Contrastive learning for multi-modal automatic code review. *arXiv preprint arXiv:2205.14289* (2022).
[76] Ayfer Ezgi Yilmaz and Haydar Demirhan. 2023. Weighted kappa measures for ordinal multi-class classification performance. *Applied Soft Computing* 134 (2023), 110020.
[77] Yongda Yu, Guoping Rong, Haifeng Shen, He Zhang, Dong Shao, Min Wang, Zhao Wei, Yong Xu, and Juhong Wang. 2024. Fine-tuning large language models to improve accuracy and comprehensibility of automated code review. *ACM transactions on software engineering and methodology* 34, 1 (2024), 1–26.
[78] Kaiyan Zhang, Yuxin Zuo, Bingxiang He, Youbang Sun, Runze Liu, Che Jiang, Yuchen Fan, Kai Tian, Guoli Jia, Pengfei Li, et al. 2025. A survey of reinforcement learning for large reasoning models. *arXiv preprint arXiv:2509.08827* (2025).
[79] Yuxin Zhang, Yuxia Zhang, Zeyu Sun, Yanjie Jiang, and Hui Liu. 2025. LAURA: Enhancing Code Review Generation with Context-Enriched Retrieval-Augmented LLM. *arXiv preprint arXiv:2512.01356* (2025).

[80] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems* 36 (2023), 46595–46623.

Received ; revised ; accepted