Review, Review, Review!

Mohamed Hedi Elfkir
Bilkent University Department of Computer Engineering
Ankara, Turkiye
hedi.elfkir@bilkent.edu.tr

Elyar Esmaeilzadeh
Bilkent University Department of Computer Engineering
Ankara, Turkiye
elyar@bilkent.edu.tr

Farzad Hallaji Azad
Bilkent University Department of Computer Engineering
Ankara, Turkiye
farzad.hallaji@bilkent.edu.tr

Ece Kunduracioglu
Bilkent University Department of Computer Engineering
Ankara, Turkiye
e.kunduracioglu@bilkent.edu.tr

Abstract

Context. Code reviews are essential for software quality, yet poorly written reviews—vague, incomplete, or verbose—impede productivity. Studies show 60% of reviews suffer from inconsistent feedback, with PR resolution times extending to 18-30 hours. While automated tools detect syntax errors and vulnerabilities, they miss nuanced human insights.

Objective. Unlike approaches that fully automate reviews or rely on static analysis, we aim to enhance human-written reviews while preserving reviewer intent and domain expertise, ensuring comprehensiveness, relevance, and conciseness.

Method. We propose an iterative refinement framework using CRScore to assess review quality across three dimensions (comprehensiveness, relevance, conciseness). Reviews below a quality threshold are iteratively refined by an LLM guided by CRScore feedback until criteria are met.

Results. Our method creates a human-AI feedback loop producing higher-quality reviews while preserving human insights. The framework can integrate as a GitHub bot for real-time assistance. Evaluation is planned using RevHelper, ChromiumConversations, and OpenDev datasets.

Keywords: Code Review, Review Quality, Human-in-the-Loop, LLMs, CRScore

ACM Reference Format:
Mohamed Hedi Elfkir, Elyar Esmaeilzadeh, Farzad Hallaji Azad, and Ece Kunduracioglu. 2018. Review, Review, Review!. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Code reviews are a cornerstone of modern software development, enabling teams to identify defects, ensure adherence to best practices, and foster knowledge sharing among developers. In collaborative environments, such as open-source projects and enterprise teams, pull requests (PRs) often undergo rigorous scrutiny to maintain code quality and prevent the introduction of bugs or suboptimal designs. However, the effectiveness of code reviews heavily depends on their quality: well-crafted reviews provide clear, actionable feedback that accelerates development cycles and reduces rework, while poorly written ones—characterized by vagueness, incompleteness, or irrelevance—can lead to misunderstandings, delays, and decreased productivity.

Traditional manual code reviews, while invaluable for capturing nuanced insights like architectural flaws or domain-specific logic errors, suffer from several limitations. Reviewers may experience fatigue, leading to overlooked issues or inconsistent feedback; scalability becomes a challenge in large teams or high-velocity projects; and the process is time-intensive, with PRs often taking 18–30 hours to resolve [5, 16]. Automated code review tools address some of these gaps by offering speed, consistency, and comprehensive coverage—analyzing thousands of lines in minutes, detecting syntax errors, common vulnerabilities (e.g., SQL injection, XSS), and code smells without bias [1]. Yet, automation alone falls short in handling complex, context-dependent problems that require human expertise, such as design decisions or edge-case behaviors.

To bridge this divide, recent advancements have explored hybrid approaches, including the use of large language models (LLMs) for generating or augmenting reviews [8]. However, directly generating reviews via LLMs risks discarding valuable human perspectives and may produce generic outputs lacking project-specific context. In this paper, we propose an iterative refinement framework that enhances human-written code reviews while preserving the reviewer's intent. Our approach leverages CRScore, an automated quality assessment tool that evaluates reviews across three key dimensions: *comprehensiveness* (coverage of relevant issues), *relevance* (focus on pertinent aspects), and *conciseness* (efficient communication without redundancy). If a review falls below a predefined threshold, it is iteratively refined by an LLM, incorporating CRScore's feedback to address deficiencies—such as missing edge cases, verbosity, or overlooked best practices—until the quality criteria are met or a maximum iteration limit is reached.

This study addresses the following research questions:

(1) **RQ1:** To what extent can iterative LLM refinement, guided by CRScore, improve the comprehensiveness, relevance, and conciseness of human-written code reviews?
(2) **RQ2:** How does the proposed refinement pipeline preserve the original intent and domain-specific insights of human reviewers?

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

This method creates a symbiotic feedback loop between human expertise and automated enhancement, resulting in higher-quality reviews that align with professional standards and improve developer efficiency. Our contributions include: (1) a novel pipeline for iterative code review refinement grounded in verifiable quality metrics; (2) empirical evaluation on datasets such as CodeReviewer [7]; and (3) potential integration as a GitHub bot for real-time review assistance.

The remainder of this paper is organized as follows: Section 2 details the problem and motivation; Section 3 reviews related work; and Section 4 describes our methodology.

2 Problem Description and Motivation

Code reviews are integral to the software development lifecycle, yet their quality remains a persistent challenge. Poorly written reviews—often vague, incomplete, or overly verbose—impede developer productivity by causing confusion, necessitating multiple revision cycles, and delaying pull request (PR) merges. Studies indicate that inconsistent feedback affects up to 60% of reviews, while manual processes can extend PR resolution times to 18–30 hours due to reviewer fatigue and scalability issues [5]. Automated tools excel at detecting syntax errors and common vulnerabilities but overlook complex logic flaws or architectural concerns that demand human insight [15]. This disparity results in suboptimal code quality, increased rework, and frustrated teams, motivating a hybrid approach that refines human reviews without supplanting them.

The core research problem we address is: *How can we systematically enhance the quality of human-written code reviews to ensure comprehensiveness, relevance, and conciseness while preserving the reviewer's original intent and domain expertise?* Current solutions either generate reviews de novo via LLMs, risking loss of nuanced human perspectives, or rely solely on static analysis, missing contextual depth. Our focus is on iterative augmentation of existing reviews, creating actionable outputs that accelerate development without introducing generic artifacts.

To illustrate what we mean by a bad review and a good review, consider the following examples.

Example 1: Bad Review

Input: A PR diff and human review comment shown below:

Human review comment: we'll need to delete this whole file, right?

```
@@ -92,7 +92,7 @@ namespace System.Diagnostics.Metrics
             T delta,
             params KeyValuePair<string, object?>[] tags)
         {
-            this.RecordMeasurement(delta, tags);
+            this.RecordMeasurement(delta, new
                 ReadOnlySpan<KeyValuePair<string, object?>>(tags));
         }
     }
 }
```

This is irrelevant and vague, ignoring the specific change.

Output: Refined review: "The change to use ReadOnlySpan for tags improves efficiency by avoiding array copies. However, ensure compatibility with older callers. Add unit test: Assert.Equal(expected, instrument.RecordMeasurement(delta, tags));. No need to delete the file; this is a minor optimization."

Example 2: Good Review

Input: A PR diff and human review comment shown below:

Human review comment: I like when `@deprecated` contains how to resolve it - eg. "use `SplFileObject::fgetcsv()` instead". The information in upgrade notes is usefull only during upgrade.

```
@@ -4,6 +4,9 @@ namespace Shopsys\FrameworkBundle\Component\Csv;
 use Symfony\Component\Filesystem\Exception\FileNotFoundException;
+/**
+ * @deprecated Class is obsolete and will be removed in the next
      major. Use SplFileObject::fgetcsv() instead.
+ */
 class CsvReader
 {
     /**
```

This is comprehensive and relevant, providing actionable advice.

Output: Because the review is already good enough, our method would not make any changes to the review.

These examples highlight how our approach transforms low-quality inputs into precise, developer-friendly outputs, directly tackling quality gaps without algorithmic details (deferred to Section 4).

3 Related Work

Our method enhances human-written code review comments by using their quality assessments as guidance for improvement. Considering the problem, our work connects two major research directions: (1) evaluating the quality of code review comments, and (2) generating high-quality code review comments. By considering these two related yet distinct perspectives in the following subsections, we first examine the metrics used to assess the quality of code review comments, including the evaluation strategies employed by comment-generation models. We then discuss prior work focused on generating code review comments. Although our goal is to enhance human-written comments rather than generate reviews entirely from scratch, our method still produces an improved version of the original comment by using its quality assessment as guidance. Thus, the methods used for generating code review comments are highly related to our approach.

3.1 Code Review Quality Assessment and Evaluation

Code reviews are an essential phase of the software development lifecycle; however, peer reviews often contain a significant number of non-useful comments [13]. Therefore, a good metric is essential to assess the quality of a given review comment. However, since there is no single correct ground truth code review, evaluating the quality of review comments is a challenging task. For example, two very different reviews may highlight different aspects of a pull request and still provide valuable insights. Some previous studies, including recent review comment-generation works [6, 7], have relied on reference-based evaluation methods [11]. These approaches typically measure the similarity between the generated comment and a human-written reference. BLEU[12], one of the most widely

used automatic evaluation metrics in natural language generation, measures n-gram overlap between a generated text and one or more human-written references using modified n-gram precision and a brevity penalty. Although BLEU was originally designed for machine translation, several review-comment generation papers have adopted it as a default evaluation metric [8]. ROUGE [9] is another reference-based metric that compares a model-generated summary with one or more human-written reference summaries by measuring lexical overlap. However, as discussed in both CRScore[11] and DeepCRCEval [10], such text-similarity metrics are limited in their ability to evaluate the actual quality of a generated review comment. They focus primarily on surface-level overlap, which does not necessarily reflect usefulness, correctness, or actionable insight. However, such reference-based criteria are not well-suited for assessing the real quality of a review. Due to these limitations, in our code review enhancement task, we adopt a reference-free evaluation approach that assesses the quality of a comment based on its inherent properties, independent of any reference text.

RevHelper [13] is one of the early works that assesses the quality of code reviews by examining their usefulness. They proposed a prediction model designed to estimate the usefulness of a code review comment before it is submitted. The authors train classical machine-learning models (Naive Bayes, Random Forest, and Logistic Regression) using a combination of textual and reviewer-related features. The textual features include reading ease, stop-word ratio, question ratio, code-element ratio, and conceptual similarity, while the developer-experience features capture aspects such as code authorship, prior review activity, and external library experience. Similar to RevHelper, our work also aims to assist developers during the review process; however, instead of merely predicting the usefulness of a comment, we take this one step further by generating an enhanced version of the human-written review based on its quality assessment.

DeepCRCEval [10] is another work that works on generating reference-free evaluation for review quality. Its authors highlighted that reference-based evaluation methods used in prior work—such as measuring text similarity—are ineffective for truly assessing the quality of code review comments. They additionally point out that existing datasets contain very limited high-quality ground-truth comments and emphasize that code review is fundamentally a defect-detection task rather than a natural language generation task. To address this, they propose an LLM-based evaluation framework, supported by human annotators, that evaluates comments across nine dimensions, including defect identification, clarity, and completeness. Their evaluation process begins with human reviewers scoring a subset of comments; these scores are then used to calibrate and validate an LLM evaluator, which continues scoring the remaining dataset. Additionally, they introduce LLM-Reviewer to demonstrate that, using few-shot learning, LLMs can generate higher-quality review comments than existing CRCG models. In contrast, our work aims to incorporate human developers directly into the comment generation process rather than focusing solely on automated quality measurement.

In our work, we utilize CRScore [11] as the quality assessment metric. It proposes a reference-free method for evaluating the quality of code review comments, eliminating the need for ground-truth comments or labeled datasets. Given a code change, the method first constructs a set of pseudo-references. These include low-level claims and high-level implications. Low-level claims are extracted from code differences using symbolic detectors, such as AST analysis, code-smell checkers, and structural change detectors. High-level implications are generated by a fine-tuned Magicoder [17] model, a decoder-only LLM, trained on synthetic GPT–4–generated claims derived from the CodeReviewer [8] dataset. After obtaining both low- and high-level pseudo-references, CRScore embeds them together with the review comment and computes pairwise Semantic Textual Similarity (STS). The resulting similarity scores are aggregated into three continuous quality metrics, such as relevance, comprehensiveness, and conciseness, each ranging from 0 to 1.

3.2 Code Review Generation

Code review comment generation aims to autonomously generate review comments by examining the pull request's code changes. Previous works have proposed different methods for generating high-quality natural language comments that identify issues in the code or suggest improvements.

In AUGER [6], the authors aimed to completely discard the human factor from the code review process, stating that generating useful review comments requires a colossal and time-consuming effort. AUGER proposes an automatic review comment generation framework based on pre-trained Transformer models (T5). It collects over 10,000 real review instances from 11 large open-source Java projects and formulates code review as a text-to-text generation task, where the model learns to map code changes and tagged lines to corresponding human-written review comments. AUGER pre-trains the T5 model to explicitly learn the relationship between a review comment and the specific part of the code that triggered it. To evaluate the generated reviews, the authors used the ROUGE metric and the Perfect Prediction rate.

CodeReviewer[7] is a framework designed for automatically generating code review comments. The authors argue that a review model must understand both code changes and review comments within the same pre-training context. To achieve this, they collect a large-scale dataset of code diffs paired with human-written review comments and pre-train a Transformer encoder–decoder model on several code-review–specific tasks. These pre-training tasks include predicting tags for code changes, reconstructing corrupted code diffs, reconstructing corrupted review comments, and generating a review comment given a code diff. The pre-trained model is then fine-tuned for downstream applications such as code change quality estimation, review comment generation, and code refinement. To evaluate the generated comments, the authors use BLEU scores and human evaluation, assessing both how informative the comment is and how well it aligns with the corresponding code change.

CRScore++ [4] is an extension of CRScore that extends the focus from evaluating review comments to training an LLM to generate higher-quality code review comments using reinforcement learning. The framework consists of two stages. In the first stage, a stronger teacher LLM generates high-quality review comments, and a smaller student model is then learn to imitate these demonstrations. In the second stage, the teacher model evaluates the student's multiple generated candidates using CRScore's quality

Table 1: A comparison table related to our work on code review comment evaluation and generation tasks

Paper	Task Focus	Generation Method	Evaluation Strategy
AUGER [6]	Automated code review comment generation	Pretrain the T5 model to learn the relationship between review comment and specific part of the code	ROUGE and Perfect Prediction rate
CodeReviewer [7]	Automating code review process	Pre-trains a Transformer model on code changes and review comments, then fine-tunes it to automatically generate code reviews	BLEU and human evaluation
DeepCRCEval [10]	Evaluating generated comments and without human interaction generating high quality comments	Human and LLM evaluators with 9 dimension scoring	LLM-Reviewer - Generating automated reviews with Few-shot prompting
CRScore [11]	Evaluation of code review comment quality	None (no comment generation)	Reference-free evaluation using low-level symbolic claims and high-level LLM-generated implications as pseudo-references
CRScore++ [4]	Generating high-quality code review comments	Teacher–student training of LLMs via reinforcement learning	LLM is used with CRScore metrics
Ours	Generating enhanced code review comments using the original human-written comment together with its quality assessment	LLM-based few-shot prompting	Reference free categorized evaluation

categories: relevance, comprehensiveness, and conciseness. The resulting preferences are used to further improve the student model through preference-based reinforcement learning. Compared with prior work, CRScore++ is the closest to ours in idea, as it also utilizes review comment qualities to improve review generation. However, unlike our method, CRScore++ focuses on generating comments entirely from scratch rather than enhancing or refining human-written comments.

In Cihan et al.'s work [2], the authors analyzed the reliability and accuracy of different LLMs, such as GPT-4o and Gemini 2.0 Flash, in code review tasks. They asked the LLMs to generate review comments that assess code correctness and suggest improvements when necessary. Their results show that GPT-4o and Gemini correctly classified code correctness 68.50% and 63.89% of the time, and produced correct code fixes 67.83% and 54.26% of the time, respectively. Based on these findings, the authors concluded that LLMs can be used in the code review process; however, due to the risk of incorrect outputs, they recommend a human-in-the-loop approach. Their proposed "Human-in-the-loop LLM Code Review" method involves an LLM reviewing all change requests, with human developers deciding whether additional human review is needed. Considering the findings of this study, we also include humans in the reviewing process; however, we give human developers a more active role by not generating comments from scratch, but instead improving and enhancing their existing comments using quality assessments.

In [3], the authors argue that LLM-based automated code review generation on its own is often unreliable, inconsistent, and prone to missing deeper semantic defects. To address this, they propose a hybrid solution that combines Knowledge-Based Systems (KBS), i.e., static analyzers that systematically follow predefined rules to detect code issues and provide reliable, structured feedback, with Learning-Based Systems (LBS), i.e., language models trained on historical review data that can recognize intricate patterns and generate contextually relevant review comments. They explore several strategies for combining these two sources of feedback, including data-augmented fine-tuning, retrieval-augmented prompting, and naive concatenation of static analyzer output with LLM-generated comments. The generated reviews are evaluated using both human annotators and LLM-as-a-judge settings.

3.3 Position of Our Work in Literature

In the software development lifecycle, code reviews play a crucial role in developing high-quality software. However, the current process, which relies solely on developers, is both time-consuming and prone to producing non-useful review comments. To improve the code review process, several works have proposed helpful tools that either assist in understanding code quality [13] or attempt to automatically generate review comments [4, 6, 8].

RevHelper [13], which is closely related to our work, provides support to reviewers by predicting whether a review comment is useful or not. We take this idea one step further by generating an enhanced version of the original human-written review rather than only evaluating it. Other studies have explored fully autonomous code review generation. However, we believe that a fully automated system is still insufficient. As highlighted in previous works [2, 3], fully automating the code review process is not entirely reliable, as LLM-based agents are prone to failures. Similar to our approach, Cihan et. al [2] suggested a human-in-the-loop review process. However, in their framework, the human serves as the evaluator of the generated review, whereas in our work, humans remain the

primary review authors, and our system enhances their comments to achieve higher quality.

In the Table 1, we summarize several methods closely related to ours, detailing their approaches to code review quality assessment as well as the techniques they employ for generating review comments.

4 Methodology

4.1 Workflow of the Proposed Method

In our method, our aim is to reach good-quality reviews while preserving the human-developer contributions that cannot be automated by AI agents. Thus, our method keeps the human reviewer in the loop and aims to enhance the original review written by the human reviewer. To better explain how our method can be integrated into the software development cycle, the activity diagram of our proposed method can be seen in the Figure. 1. After a pull request is submitted, a reviewer inspects the code changes and writes an initial review comment. This comment is then evaluated by an automated review-quality agent, CRScore, which assesses the quality of the review based on the code differences and the review text. If the quality score is below a predefined threshold, an LLM-based review enhancement module iteratively improves the comment. The module attempts to refine the review for n-iterations by using the previous quality report as guidance. Once the enhanced comment exceeds the quality threshold or reaches the maximum number of iterations, the framework provides the enhanced review as an alternative to the reviewer. In this way, our approach preserves the human reviewer's conceptual understanding of the project and the additional domain knowledge that often cannot be captured by automated AI agents. Overall, the proposed system acts as a supportive tool in the software development lifecycle, helping reviewers produce clearer and more effective code review comments.

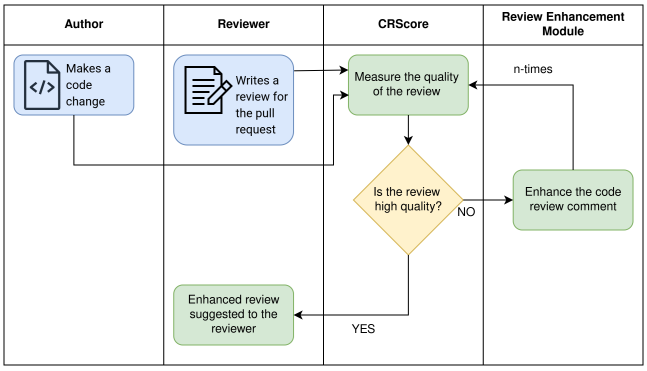

Figure 1: Activity diagram of the proposed human-in-the-loop review workflow. After the reviewer writes the initial comment, CRScore evaluates its quality. If the score meets the threshold, the review is sent to the author; otherwise, an LLM agent iteratively improves the comment until it passes the threshold or reaches the iteration limit.

4.2 Code Review Quality Assessment with CRScore

CRScore is a reference-free evaluation metric for assessing code review comment quality across three dimensions: Relevance, Comprehensiveness, and Conciseness. In our method, we are planning to use the scores produced by these dimensions as inputs to our LLM-based enhancement agent. To construct its pseudo-reference, CRScore first performs low-level claim extraction, where symbolic detectors analyze the code patch and generate factual statements describing the changes. These detectors also identify code smells, such as unused variables, unnecessary object creation, syntax issues, leaking variables, and type-conversion problems. For high-level implications, CRScore employs a fine-tuned Magicoder [17] model to infer semantic consequences of the change. Although the authors released their codebase, no pretrained Magicoder model was provided, requiring us to either reproduce their training process or use an alternative model to obtain comparable results. After constructing the pseudo-reference (combining low-level claims, high-level implications, and detected issues), CRScore embeds both the pseudo-reference and the review comment sentences using a Sentence Transformer model: Sentence-bert [14]. Finally, pairwise Semantic Textual Similarity (STS) scores—computed through cosine similarity between pseudo-reference embeddings P and review sentence embeddings R are aggregated to produce the three continuous quality metrics.

Conciseness measures the proportion of review sentences that meaningfully match at least one pseudo-reference claim/implication. If its maximum similarity exceeds a threshold τ, the sentence is considered informative.

$$\text{Con} = \frac{1}{|R|} \sum_{r \in R} \mathbb{I}\left[\max_{p \in P} \text{cosSim}(p, r) > \tau\right]. \tag{1}$$

Comprehensiveness measures the proportion of pseudo-reference items that are covered by at least one sentence in the review.

$$\text{Comp} = \frac{1}{|P|} \sum_{p \in P} \mathbb{I}\left[\max_{r \in R} \text{cosSim}(p, r) > \tau\right]. \tag{2}$$

Relevance combines conciseness and comprehensiveness via a harmonic mean, rewarding reviews that are both focused and complete.

$$\text{Rel} = \frac{2 \cdot \text{Con} \cdot \text{Comp}}{\text{Con} + \text{Comp}}. \tag{3}$$

4.3 LLM-based Review Enhancement

In the LLM-based review enhancement module, our initial approach begins with using an off-the-shelf LLM agent and applying few-shot prompting to enhance the original human reviews. For few-shot prompting, we plan to provide the model with several enhancement examples so that the agent can infer the expected transformation pattern. To construct these examples, we will obtain high-quality ground-truth reviews and systematically degrade them into low-quality variants that CRScore would score poorly. This allows the model to observe clear before-and-after improvement pairs and learn the style of enhancement we expect. If this strategy proves insufficient, we plan to train our model using a few-shot learning

approach that allows the model to better adapt to the specific characteristics of our task. As a final fallback strategy, we may fine-tune the LLM, taking our hardware limitations into account.

For individual review enhancements, we plan to use the instruction prompt provided below.

LLM Enhancement Prompt

```
You are a senior software engineer. Your task is to improve
      a junior developer's review comment.
Preserve the reviewer's original intent, avoid
     hallucinations, and ensure that the feedback you
     provide is accurate and grounded in the code change.

A quality assessment has been performed on the junior
     developer's comment.
The assessment contains three dimensions, each scored in
     the range $[0, 1]$.
Use these scores to guide the enhancement, giving more
     attention to the dimensions with lower scores.
The dimensions are:

\begin{itemize}
   \item \textbf{Comprehensiveness}: coverage of relevant
        issues,
   \item \textbf{Relevance}: focus on pertinent aspects of
        the change,
   \item \textbf{Conciseness}: clarity and efficiency
        without redundancy.
\end{itemize}

Here are the results of the quality assessment: \\
\texttt{\{insert quality scores here\}}

Here is the code change: \\
\texttt{\{insert code diff here\}}

Here is the junior developer's review comment: \\
\texttt{\{insert review comment here\}}
```

References

[1] Aikido Security. 2024. SAST Platform - Static Code Analysis. https://www.aikido.dev/scanners/static-code-analysis-sast. Accessed: 2025-11-11.

[2] Umut Cihan, Arda İçöz, Vahid Haratian, and Eray Tüzün. 2025. Evaluating Large Language Models for Code Review. *arXiv preprint arXiv:2505.20206* (2025).

[3] Imen Jaoua, Oussama Ben Sghaier, and Houari Sahraoui. 2025. Combining Large Language Models with Static Analyzers for Code Review Generation. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 174–186.

[4] Manav Nitin Kapadnis, Atharva Naik, and Carolyn Rose. 2025. CRScore++: Reinforcement Learning with Verifiable Tool and AI Feedback for Code Review. *arXiv preprint arXiv:2506.00296* (2025).

[5] Kodus. 2024. Manual Code Review vs Automated: What Really Works? https://kodus.io/en/manual-vs-automated-code-review/. Accessed: 2025-11-11.

[6] 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.

[7] 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.

[8] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 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 (ESEC/FSE 2022)*. ACM, 1035–1047. doi:10.1145/3540250.3549108

[9] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[10] Junyi Lu, Xiaojia Li, Zihan Hua, Lei Yu, Shiqi Cheng, Li Yang, Fengjun Zhang, and Chun Zuo. 2025. Deepcrceval: Revisiting the evaluation of code review comment generation. In *International Conference on Fundamental Approaches to Software Engineering*. Springer Nature Switzerland Cham, 43–64.

[11] Atharva Naik, Marcus Alenius, Daniel Fried, and Carolyn Rose. 2025. Crscore: Grounding automated evaluation of code review comments in code claims and smells. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. 9049–9076.

[12] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[13] Mohammad Masudur Rahman, Chanchal K. Roy, and Ryuji G. Kula. 2018. *Predicting Usefulness of Code Review Comments using Textual Features and Developer Experience*. https://arxiv.org/abs/1807.04485

[14] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).

[15] Sunbytes. 2024. Manual vs automated code review: Which is the best? https://sunbytes.io/blog/manual-code-review-vs-automated-code-review/. Accessed: 2025-11-11.

[16] TCM Security. 2024. Manual vs Automated Code Review. https://tcm-sec.com/manual-vs-automated-code-review/. Accessed: 2025-11-11.

[17] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Empowering code generation with oss-instruct. *arXiv preprint arXiv:2312.02120* (2023).