Retrieval-Augmented Code Review Comment Generation

Hyunsun Hong
School of Computing
KAIST
Daejeon, Republic of Korea
hyunsun.hong@kaist.ac.kr

Jongmoon Baik
School of Computing
KAIST
Daejeon, Republic of Korea
jbaik@kaist.ac.kr

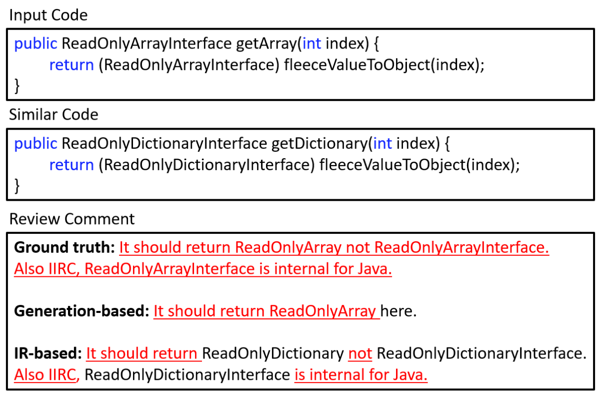
***Abstract*—Automated Code review comment generation (RCG) aims to assist developers by automatically producing natural language feedback for code changes. Existing approaches are primarily either generation-based, using pretrained language models, or information retrieval-based (IR), reusing comments from similar past examples. While generation-based methods leverage code-specific pretraining on large code–natural language corpora to learn semantic relationships between code and natural language, they often struggle to generate low-frequency but semantically important tokens due to their probabilistic nature. In contrast, IR-based methods excel at recovering such rare tokens by copying from existing examples but lack flexibility in adapting to new code contexts—for example, when input code contains identifiers or structures not found in the retrieval database. To bridge the gap between generation-based and IR-based methods, this work proposes to leverage retrieval-augmented generation (RAG) for RCG by conditioning pretrained language models on retrieved code–review exemplars. By providing relevant examples that illustrate how similar code has been previously reviewed, the model is better guided to generate accurate review comments. Our evaluation on the Tufano et al. benchmark shows that RAG-based RCG outperforms both generation-based and IR-based RCG. It achieves up to +1.67% higher exact match and +4.25% higher BLEU scores compared to generation-based RCG. It also improves the generation of low-frequency ground-truth tokens by up to 24.01%. We additionally find that performance improves as the number of retrieved exemplars increases.**

***Index Terms*—Code Review Automation, Deep Learning, Software Quality Assurance, Retrieval-Augmented Generation**

I. Introduction

Code review is a systematic and collaborative process that involves examining code written by others to assess its quality and readiness for integration. If a code change is deemed unsatisfactory, the reviewer may reject the merge request and provide feedback in the form of review comments, enabling the author to revise the code accordingly. This process plays a key role in identifying latent issues early in the development cycle, preventing potential defects, and improving maintainability through refactoring [1]. For this reason, code review is widely adopted and strongly recommended practice as a core component of software quality assurance [2]. However, code review is inherently labor-intensive and time-consuming, requiring experienced developers to thoroughly inspect changes and deliver precise, constructive feedback [3]. Studies report that developers spend an average of six hours per week on review-related activities, including reading, understanding, and providing feedback on code changes [4]. Moreover, authors often face delays of 15 to 64 hours before receiving review comments, which can hinder development velocity [5]. As software systems grow, the volume of code review requests increases significantly; for example, Microsoft Bing reportedly conducts approximately 3,000 code reviews per month [6]. These challenges have led researchers to explore automating the review process. Review Comment Generation (RCG) has emerged as a promising approach, aiming to ease the burden of manual review comment writing by automatically generating relevant, context-aware feedback for code changes.

RCG has been researched in two directions: Generation-based RCG and Information-Retrieval (IR)-based RCG. Generation-based RCG utilizes a generative language model. With the hypothesis that software exhibits repetitive and predictable property [7], generation-based RCG takes advantage of probabilistic language models trained on the large code-review corpus to generate most probable code review words. Tufano et al. [8], [9] were first to apply pre-trained language model (PLM) to code review automation and Li et al. [10] later proposed a code review-specific pre-training methodology. Both studies contributed benchmark datasets that have since become foundational for evaluating RCG methodology [9], [10]. Based on these benchmarks, subsequent works have focused on adapting advanced natural language processing (NLP) techniques to enhance generation-based RCG [11]–[13]. IR-based RCG, on the other hand, utilizes information retrieval. It finds the most probable code reviews from a large code review history database and regards these retrieved exemplars as code reviews for the input code [14]–[16]. Rather than generating new text from generative language model, IR-based RCG reuses exemplar mined from a code review history database. It is more efficient than generation-based RCG because it does not require any training [14]. Notably, CommentFinder [14], an IR-based RCG approach, reports a 32% performance improvement over the generation-based method proposed by Tufano et al. [9] in RCG. This enhancement is achieved with a 49× speedup over the generation-based method [9], by leveraging simple Bag-of-Words lexical similarity to retrieve similar code examples from the training dataset. Although both generation-based RCG and IR-based


Input Code

```
public ReadOnlyArrayInterface getArray(int index) {
      return (ReadOnlyArrayInterface) fleeceValueToObject(index);
}
```

Similar Code

```
public ReadOnlyDictionaryInterface getDictionary(int index) {
      return (ReadOnlyDictionaryInterface) fleeceValueToObject(index);
}
```

Review Comment

Ground truth: It should return ReadOnlyArray not ReadOnlyArrayInterface. Also IIRC, ReadOnlyArrayInterface is internal for Java.

Generation-based: It should return ReadOnlyArray here.

IR-based: It should return ReadOnlyDictionary not ReadOnlyDictionaryInterface. Also IIRC, ReadOnlyDictionaryInterface is internal for Java.

Fig. 1. A motivating example of generation-based and IR-based code review comment generation. Words in ground truth were underlined.

RCG have been studied independently with its own advantages, their integration has not been explored. However, there is room for two methods to be integrated harmoniously.

As generative language models innately tend to generate high-frequency words, it has trouble with generating low-frequency but semantically important tokens—referred to as Low-Frequency Ground-Truth Tokens (LFGTs) [17], [18]. Accurately generating LFGTs is critical for improving the overall quality and usefulness of generated review comments. We hypothesize that incorporating retrieval candidates from IR-based RCG can expose the generative model to informative exemplars containing LFGTs, thereby guiding the language model to generate contextually appropriate LFGTs.

Figure 1 shows a motivating example. This is an example result of generation-based RCG (CodeT5 [19]) and IR-based (CommentFinder [14]) RCG on the Tufano et al.'s benchmark dataset [9]. Given an input code snippet, the generation-based approach produces review comments using its parametric knowledge acquired during training. In contrast, the IR-based method identifies lexically similar code from the training dataset and directly reuses the associated review comment as the output. The generation-based model tends to produce high-frequency tokens such as "it" (46,654 occurrences in the training corpus), "should" (26,721), and "return" (142,643), but it fails to generate LFGTs like "Also" (7,476), "IIRC" (123), "internal" (1,621), and "Java" (7,016). Although the model successfully generate "ReadOnlyArray" (1), a token that appears only once in the training corpus, this is primarily due to its presence in the input code itself, rather than the model's ability to generalize to rare vocabulary. Conversely, the IR-based method is capable of retrieving review comments that contain LFGTs such as "Also," "IIRC," "internal," and "Java" by leveraging similar examples in the retrieval set. However, it fails to retrieve identifier-specific tokens like "ReadOnlyArray" when such tokens are rare in the retrieval corpus. This example illustrates a complementary gap between generation-based RCG and IR-based RCG. While generation-based RCG can effectively generate LFGTs that are present in the input code (e.g., "ReadOnlyArray"), it struggles to produce LFGTs that are absent from the input. Conversely, IR-based RCG can retrieve LFGTs from similar examples in the retrieval corpus even when they are not present in the input code (e.g., "IIRC" or "Java"), but it struggles to capture identifier tokens that appear in the input but are rare in the retrieval set. This observation motivates our investigation into a hybrid approach that integrates both paradigms to leverage their respective strengths.

To bridge the complementary gap between generation-based and IR-based RCG, we propose to leverage Retrieval Augmented Generation (RAG) [20] to RCG by augmenting input code query with top-k most similar code's review exemplars. RAG enhances generative language model's performance on knowledge-intensive tasks by augmenting input queries with relevant knowledge passages [20]. By incorporating retrieved exemplars, RAG enables generative language models to access domain-specific knowledge and refer to LFGTs without requiring additional training [21]. Building on this principle, RAG has been effectively employed in diverse software engineering tasks, including code summarization [22], code completion [23], code generation [24], and test case generation [25]. Given that code review requires extensive knowledge of project-specific practices—such as coding conventions, review focus areas, and stylistic guidelines [26]—RAG can be beneficial, as retrieved exemplars help convey this domain-specific knowledge to the generation model. However, the use of RAG for generating review comments has not been explored. To address this gap, we propose RAG-Reviewer: RAG-based RCG framework that integrates IR-based and generation-based approaches.

The RAG-Reviewer framework comprises two primary components: a retrieval module and a generation module. The retrieval module retrieves top-k similar review exemplars that are most similar to the input code snippet from a large historical code review database. The generation module utilizes a PLM to generate a review comments for the input code snippet with the retrieved exemplars. Given an input code for review, the retrieval module first identifies the top-k most similar code examples and extracts their corresponding review comments. These retrieved exemplars are then concatenated with the input code and passed to the generation module. To adapt the generator for this retrieval-augmented input format, we fine-tune the language model to produce review comments that jointly reflect information from both the input and the retrieved examples. Empirical evaluations demonstrate that RAG-Reviewer outperforms both generation-only and IR-only baselines. It achieves consistent improvements in Exact Match and BLEU scores across all PLMs and enhances the generation of LFGTs. The key contributions of this work are as follows:

- We propose **RAG-Reviewer**, a RAG framework for RCG that unifies generation-based and IR-based approaches. To the best of our knowledge, this is the first study to apply the RAG paradigm to the RCG task.
- We conduct comprehensive experiments across multiple PLMs, showing that RAG-Reviewer improves generation accuracy, better captures LFGTs, and exhibits perfor-

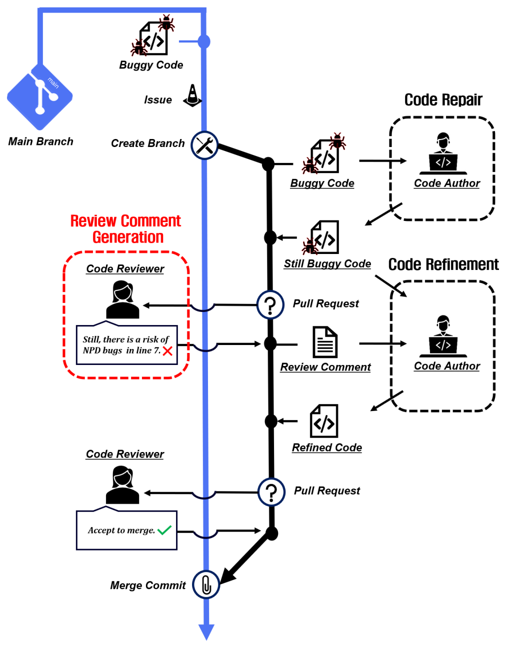


Fig. 2. Code review process workflow

mance improvements as the number of retrieved exemplars increases.

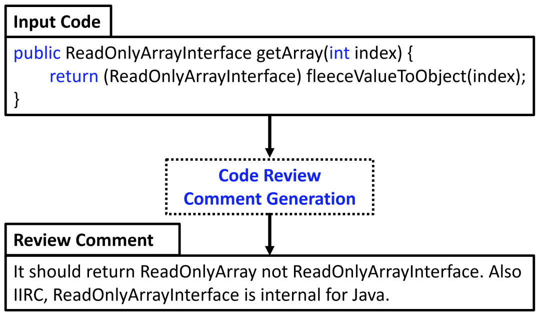
- To support reproducibility and encourage future research, we publicly release our implementation on GitHub: https://github.com/RAG-Reviewer/RAG-Reviewer.

II. Background

In this section, we provide a brief overview of code review process and code review comment generation.

A. Code Review Process

Code review is the one of the widely adapted software quality assurance practices in both open-source projects and industries [27], [28]. Through code review process, various aspects of code quality (e.g., readability, maintainability, testability, scalability, secure and more) can be inspected by collective expertise of reviewers. Figure 2 shows code review process scenario in debugging the faulty code. Suppose that an issue occurred in main branch due to a bug. Then, (1) the author who is responsible for this issue repairs the buggy code and requests the code reviewer to pull repaired code into main branch. Later, (2) the code reviewer evaluates the proposed code to determine whether the revised code is suitable for integration into the main branch. If the code still contains defects, the reviewer rejects the pull request and provides feedback through a review comment. (3) The author subsequently revises the code based on the reviewer's suggestions and resubmits the pull request. (4) Upon re-evaluation, if the changes address the identified issues, the reviewer approves the code for merging into the main branch. Through this code review process, all codes can be inspected by peer reviewers before it integrated to the main codebase, thereby enhancing overall software quality.



```
public ReadOnlyArrayInterface getArray(int index) {
      return (ReadOnlyArrayInterface) fleeceValueToObject(index);
}
```

It should return ReadOnlyArray not ReadOnlyArrayInterface. Also IIRC, ReadOnlyArrayInterface is internal for Java.

Fig. 3. An example of code review comment generation

Within this process, code review process involves three tasks: code repair (code-to-code), review comment generation (code-to-comment) and code refinement (code&comment-to-code) as illustrated in Figure 2. The field of Code Review Automation (CRA) aims to automate these three tasks [9] and we focus on automated review comment generation in this work.

B. Code Review Comment Generation

Code Review Comment Generation (RCG) automates the generation of review comments in the code review process, as shown in Figure 3. RCG assists code reviewers by automatically generating review comments, which can serve as references or starting points for writing their own final feedback. Formally, given an input code sequence to be reviewed, $C_{input} = \{c_1, c_2, \ldots, c_n\}$, and its corresponding code review comments, $N_{Review} = \{n_1, n_2, \ldots, n_m\}$, generation-based RCG is a sequence-to-sequence probabilistic mapping function $G : C_{input} \rightarrow N_{Review}$. In a probabilistic sequence modeling framework, the probability of generating N_{Review} given C_{input} is defined as:

$$P_G(N_{Review} \mid C_{input}) = \prod_{i=1}^{m} P_G(n_i \mid C_{input}, n_1, \ldots, n_{i-1})$$

where P_G denotes a probabilistic autoregressive model parameterized by G.

III. Approach

A. Overall Approach

In this work, we propose **RAG-Reviewer**, a **RAG**-based code **Review** comment gen**er**ation framework. RAG-Reviewer integrates the strengths of both IR-based and generation-based methods. It consists of two core components: (a) **Retrieval Module** that retrieves review comments corresponding to code snippets that are most similar to the given input code and (b) **Generator Module** that produces review comments for the input code by conditioning on both the input and the retrieved exemplars.

By augmenting the input code with exemplars retrieved via the retrieval module, the generative language model can leverage the outputs of the IR-based approach to enhance the quality of generated review comments. Figure 4 provides an overview of RAG-Reviewer, which operates in two phases:

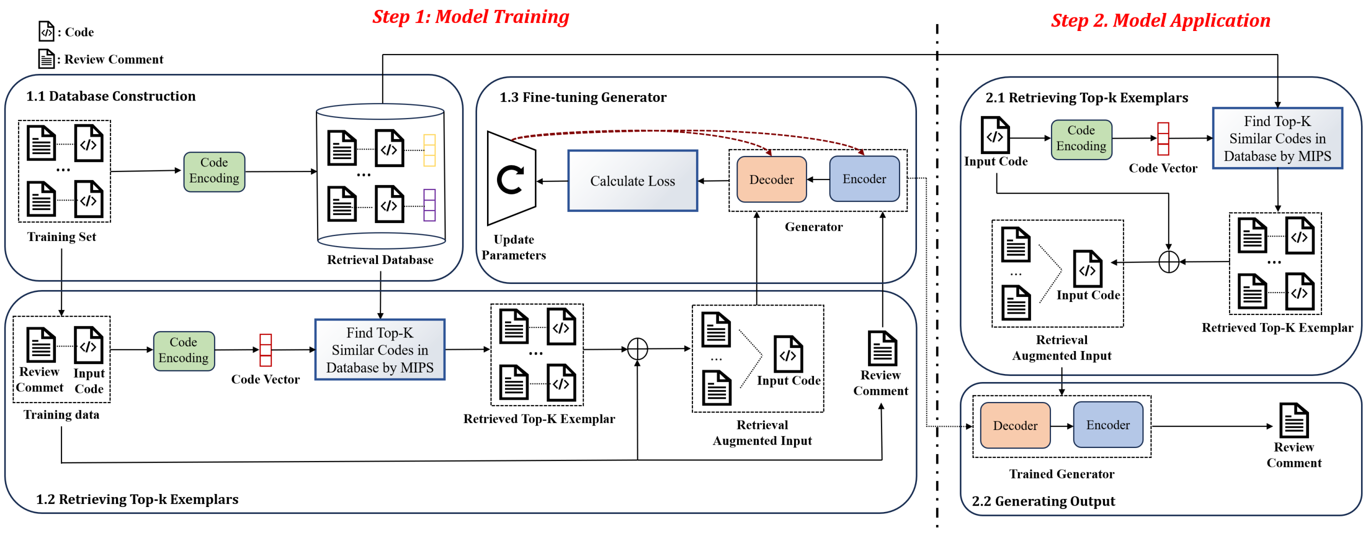


Fig. 4. Overall framework of RAG-Reviewer

model training and model application. During the training phase, RAG-Reviewer constructs a retrieval database from the training dataset, comprising code snippets, their corresponding review comments, and their encoded vector representations. Each code snippet is encoded into a dense vector space to facilitate similarity computation using Maximum Inner Product Search (MIPS). The retrieval module then identifies the top-k most similar code snippets from this database using the same encoding mechanism. To prevent data leakage, the top-1 retrieved exemplar—which is identical to the input training instance—is excluded during training. The generator is fine-tuned on these augmented inputs, learning to condition on both the input code and the retrieved exemplars. During model application, the same retrieval procedure is applied to obtain relevant exemplars for a new input code snippet.

B. Retrieval Module

The retrieval module is responsible for identifying the most relevant exemplars from the retrieval database given an input code snippet. These retrieved exemplars provide contextual guidance that the generator can leverage to produce more accurate and informative review comments. The retrieval database is constructed from the training dataset. Formally, the training dataset is defined as a set $T_{\text{review}} = \{(IC_i, CR_i) | i = 1, \ldots, M\}$ where IC_i denotes the input code snippet, CR_i represents its corresponding ground-truth review comment and M is the total number of training instances. This dataset serves as the basis for both retrieval and generator training within the RAG-Reviewer framework. To enable similarity computation between the input code and the code snippets in the retrieval database, all input code instances IC_is must be encoded into dense vector representations. This vectorization allows for the efficient measurement of semantic similarity—typically using cosine similarity—between code snippets. Formally, we employ an encoder d_c:

$$h_i = d_c(IC_i), \forall (IC_i, CR_i) \in T_{\text{review}} \tag{1}$$

where h_i is the vectorized representation of IC_i. We use the UniXcoder [29] as d_c, because it can effectively preserve code's semantic and syntactic information. Especially, UniXcoder showed state-of-the-art performance in IR-based RCG [16]. Based on this, we construct the retrieval database D_{review} as follows:

$$D_{\text{review}} = \{(h_i, IC_i, CR_i) | i = 1, \ldots, M\} \tag{2}$$

To find the most similar exemplars corresponding to a given input code C_{input}, we encode C_{input} into h_x using d_c:

$$h_x = d_c(C_{\text{input}}) \tag{3}$$

Note that we used same encoder d_c for both C_{input} and IC_i because they are both forms of code, making them unimodal. Using the same encoder ensures consistency in the dense vector representation. After encoding, we compute similarity score s_i between C_{input} and IC_is using the inner dot product:

$$s_i = \langle h_x, h_i \rangle, \forall i \in \{1, \ldots, M\} \tag{4}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product. Based on the computed similarity scores, we then select the top-k most similar exemplars with respect to the input code:

$$\mathcal{C}_R = \text{TopK}\left(\{s_i\}_{i=1}^{M}, \mathcal{D}_{\text{review}}, K\right) = \{(IC_1, CR_1), \ldots, (IC_k, CR_k)\} \tag{5}$$

where $\text{TopK}(S, D, K)$ returns top-k most similar code review exemplars based on the similarity score set S, retrieval database D and specified value of K. The resulting top-k retrieval candidates C_R are then used to augment the input in the generation module.

C. Generation Module

Given the output (Equation 5) of retrieval module, the list of top-k most similar exemplars with respect to the input code, we construct the input to the generator by concatenating the input code sequence X with the retrieved candidates C_R:

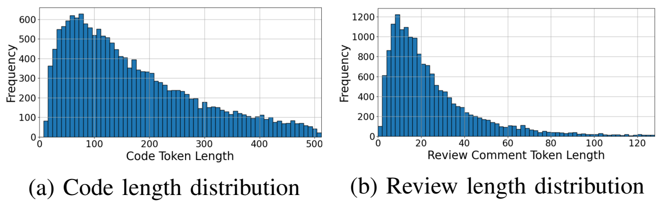
$$X' = X \oplus C_1 \oplus C_2 \cdots \oplus C_k, \quad C_i \in C_R \tag{6}$$

TABLE I
FREQUENCY DISTRIBUTION OF REVIEW COMMENTS IN TUFANO ET AL.'S TRAINING DATASET

Token Freq. ($\leq x$)	≤ 1	≤ 5	≤ 10	≤ 20	≤ 50	≤ 100	All (≥ 1)
Unique Tokens Count	2,592 (10.08%)	8,827 (34.23%)	12,817 (49.84%)	16,558 (64.39%)	20,544 (79.88%)	22,507 (87.52%)	25,717 (100%)
# Review Comments Containing Tokens	2,486 (1.85%)	16,944 (12.62%)	32,768 (24.41%)	52,597 (39.18%)	80,620 (60.06%)	98,550 (73.41%)	134,239 (100%)

TABLE II
STATISTICS OF TUFANO ET AL.'S DATASET

Dataset	**Train**	**Valid**	**Test**
Count	134,239	16,780	16,780
Avg. Code Length	167	168	171
Avg. Review Length	26	26	26



(a) Code length distribution (b) Review length distribution

Fig. 5. Token length distribution of Tufano et al.'s test dataset

where C_i denotes i^{th} most similar code's exemplar. Each C_i consists of a code snippet IC_i and its associated review comment CR_i. Thus, we have to decide what to retrieve for each C_i. Following methodology of Parvez et al. [24], we explore two input augmentation strategies, **pairs** and **singleton**. The pair setting incorporates both the exemplar code and its corresponding review comment, whereas the singleton setting leverages only the review comment:

$$X'_p = X \oplus CR_1 \oplus IC_1 \cdots, \quad (IC_i, CR_i) \in \mathcal{C}_R \tag{7}$$
$$X'_s = X \oplus CR_1 \oplus CR_2 \cdots, (IC_i, CR_i) \in \mathcal{C}_R \tag{8}$$

where IC_i means i^{th} most similar code and CR_i means its corresponding review comment retrieved from database D_{review}. To clearly delineate the boundaries between code and review comments within each exemplar, we introduce special delimiter tokens, [csep] and [nsep], following the input formatting strategy proposed by Parvez et al. [24].

Given the augmented input X', RAG-Reviewer generate the output sequence $Y = (y_1, \ldots, y_N)$ using a generative language model under an autoregressive formulation:

$$P_\theta(Y \mid X') = \prod_{i=1}^{N} P_\theta(y_i \mid X', y_{1:i-1}) \tag{9}$$

where P_θ is a language model parameterized by θ, which generates each token conditioned on the input X' and previous $i-1$ tokens. We experiment with various types of pre-trained language models as the generator and compare their performance within the RAG-Reviewer framework.

To train the generator, RAG-Reviewer minimizes the following negative log-likelihood loss function over the target sequence $Y^{\text{true}} = (y_1^{\text{true}}, \ldots, y_N^{\text{true}})$:

$$L(\theta) = -\sum_{i=1}^{N} \log P_\theta(y_i^{\text{true}} \mid X', y_{1:i-1}^{\text{true}}) \tag{10}$$

where θ denotes the trainable parameters of the generator. We fine-tuned only the generator while keeping the code encoder d_c fixed because training d_c would necessitate updating h_i (Equation 1) at every gradient step, as changes to d_c would alter h_i, making recalculations computationally expensive [20].

IV. EXPERIMENT SETUP

A. Research Question

We formulate the following research questions to evaluate the effectiveness of RAG-Reviewer:

- **RQ1:** Does RAG-Reviewer outperform existing baseline methods in terms of review comment generation quality?
- **RQ2:** Does RAG-Reviewer mitigate the challenges faced by language models in generating low-frequency tokens?
- **RQ3:** What is the impact of the number of retrieved exemplars on the performance of RAG-Reviewer?

These research questions are empirically investigated in Section V.

B. Dataset

We use the dataset introduced by Tufano et al. [9]. which was constructed from large-scale Java open-source projects on GitHub and Gerrit. Each code is function-level granularity written in Java and paired with its corresponding review comment. Detailed statistics are shown in Table II, and Figure 5 illustrates the token length distribution in the test set. All lengths were calculated using the CodeT5 [19] tokenizer. As shown in Table I, 87.52% of unique tokens in the review comment training corpus occur no more than 100 times, and 73.41% of comments contain at least one such token. Following prior work [22], [30], we define these as low-frequency tokens. This high prevalence makes the Tufano et al. dataset a suitable benchmark for evaluating our approach to improving low-frequency token generation.

C. Evaluation Metrics

Following the previous code review comment generation work [9]–[11], [14], we evaluate the quality of generated review comments using the metric BLEU [31] and Exact Match (EM) score. These metrics are widely used in other automated software engineering work [22]–[24], [30].

BLEU measures the frequency of overlapped n-grams between the generated comment and the reference comment using n-gram precision. We use BLEU-4, which considers up to 4-gram matches. The BLEU-N score is computed as:

$$\text{BLEU-N} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

where p_n is the n-gram precision, w_n is the weight for each n-gram (uniformly set to $1/N$), and BP is a brevity penalty that penalizes overly short generated sequences. Higher BLEU scores indicate greater similarity to the ground truth review comment, ranging from 0% to 100%.

EM measures the ratio of generated outputs that are exactly identical to the ground truth review comment. It is a stricter metric than BLEU because it requires a character-wise match. The EM score also ranges from 0% to 100%, where 100% indicates that all the generated outputs for the test dataset are character-wise identical to the ground truth.

D. Baselines

To evaluate the effectiveness of **RAG-Reviewer**, we compare it against a range of baselines, including both generation-based and IR-based methods. We reproduced all the baselines.

- **CommentFinder**: A first IR-based RCG approach that uses Bag-of-Words (sparse vector) for code vectorization and cosine similarity for candidate ranking [14].
- **UniXCoder-IR**: Proposed by Kartal et al. [16], this method uses UniXCoder (dense vector) [29] for code vectorization. Retrieval is based on similarity metrics such as cosine or Euclidean distance.
- **Tufano T5**: A small version of the T5 [32] architecture (6-layer encoder and 6-layer decoder) pre-trained for code review automation. It was pre-trained on code and natural language pairs collected from Stack Overflow and CodeSearchNet [33] using masked language modeling [9].
- **CodeT5**: A pre-trained encoder-decoder model that supports both code understanding and generation. Based on the T5 [32] architecture, CodeT5 [19] introduces identifier-aware objectives and bimodal dual generation tasks. It was pre-trained on CodeSearchNet [33] and additional GitHub repositories. We use the `CodeT5-base` variant, which has 12 encoder and 12 decoder layers.
- **CodeT5+**: An extended version of CodeT5 that incorporates a mixture of pre-training objectives including span denoising, contrastive learning, text–code matching, and causal language modeling. CodeT5+ [34] is trained on both unimodal code and bimodal code–text corpora. We use the `CodeT5p-220m` version.
- **CodeReviewer**: Specifically pre-trained for code review automation tasks, CodeReviewer [10] leverages a multilingual dataset of code diffs and review comments from GitHub. It introduces four novel pre-training tasks tailored for code review, such as diff tag prediction and denoising objectives for both code and review comments.
- **AUGER**: AUGER [35] is a T5-based [32] model that enhances review comment generation using review-line tagging and cross pre-training. It is trained on a curated dataset of over 10,000 real-world code change and review comment pairs from popular Java projects. AUGER incorporates review-specific tags and demonstrates significant improvements in review comment quality.

E. Implementation Details

All experiments were conducted on a NVIDIA Tesla V100-SXM2-32GB GPU. For `CodeT5`, `CodeT5+` and `CodeReviewer`, we used the AdamW optimizer (learning rate = 3e-5, weight decay = 0.01) with a linear learning rate schedule and 10% warm-up steps. To address memory constraints, we applied gradient accumulation over 3 steps with a batch size of 12 (effective batch size = 36). We used gradient clipping with a maximum norm of 1.0 and enabled mixed-precision training. The input and output lengths were capped at 512 tokens for code and 128 tokens for review comments, based on the dataset's token distribution (see Table II and Figure 5). Longer input sequences were truncated to fit within token limits. For RAG-based models, input prompts were augmented with as many retrieved exemplars as would fit within the input token budget. We used a beam size of 10 for evaluation. Training was conducted for up to 20 epochs, with early stopping applied after 3 epochs of no improvement on the validation set. The best model checkpoint was selected based on validation performance. For `Tufano T5` and `AUGER`, we used the Adam optimizer (learning rate = 3e-4) with a batch size of 12 and no gradient accumulation, based on preliminary tuning results. All baseline models were implemented using publicly available replication packages [36]–[42].

V. Experimental Result

In this section, we present experimental result of our work following three research questions.

*A. **(RQ1) Does RAG-Reviewer outperform existing baseline methods in terms of review comment generation quality?***

Singleton vs. Pair Retrieval Strategy. Before comparing RAG-Reviewer with existing baselines, we first evaluate which retrieval strategy achieves superior performance within RAG-Reviewer. Following Parvez et al. [24], we consider two strategies: (1) *Singleton Retrieval*, which retrieves a larger number of review comments without their associated code snippets and (2) *Pair Retrieval*, which retrieves fewer exemplars but preserves both the code snippets and their corresponding review comments.

As shown in Table III, the pair retrieval strategy consistently outperforms singleton retrieval in terms of both EM and BLEU scores across all PLMs. The EM gains range from +0.14% (AUGER: 2.32% $\rightarrow$ 2.46%) to +0.53% (Tufano T5: 2.01% $\rightarrow$ 2.54%). The BLEU gains range from +0.53% (CodeT5: 12.45% $\rightarrow$ 12.98%) to +1.76% (CodeReviewer: 11.76% $\rightarrow$ 13.52%). These results suggest that providing both code and its review comment in retrieved exemplars helps the model better learn code–comment relationships. Given the inherent input token limitations of language models, it is crucial to include

TABLE III
PERFORMANCE COMPARISON OF RAG-REVIEWER WITH GENERATION-BASED AND IR-BASED BASELINES

Name	EM (%)	BLEU (%)
Information Retrieval		
CommentFinder	2.80	12.41
UniXCoder-IR	2.79	12.80
Generation		
Tufano T5	0.87	9.32
CodeReviewer	1.23	9.27
CodeT5	1.39	9.91
CodeT5+	1.71	11.59
AUGER	1.10	9.78
Retrieval Augmented Generation (Singleton)		
Singleton Tufano T5	2.01	11.82
Singleton CodeReviewer	2.32	11.76
Singleton CodeT5	2.40	12.45
Singleton CodeT5+	2.53	11.97
Singleton AUGER	2.32	11.81
Retrieval Augmented Generation (Pair)		
Pair Tufano T5	2.54	12.54
Pair CodeReviewer	2.76	**13.52**
Pair CodeT5	2.90	12.98
Pair CodeT5+	**3.01**	12.39
Pair AUGER	2.46	12.93

helpful information—both in quantity and quality—within the available budget. Our experiments show that rather than allocating tokens to more review-only exemplars, it is more effective to include both the review comments and their corresponding code. Notably, since code snippets are typically much longer than comments (see Table II), pair retrieval fits much fewer exemplars within the token limit. Nevertheless, providing richer contextual information through paired code-comment exemplars yields better performance than using a larger number of comment-only examples.

Generation-based vs. RAG-Reviewer. RAG-Reviewer consistently outperforms generation-based models across all PLMs in both EM and BLEU scores. Notably, even models with modest baseline performance, such as Tufano T5 and CodeReviewer, show the most significant gains—Tufano T5 improves from 0.87% to 2.54% in EM (+1.67%), while CodeReviewer improves from 9.27% to 13.52% in BLEU (+4.25%). These results suggest that retrieval augmentation benefits not only strong backbones but also weaker ones, enhancing both precision and fluency by grounding generation in relevant contextual exemplars.

IR-based vs. RAG-Reviewer. Compared to IR-based methods, RAG-Reviewer achieves comparable or slightly higher performance in most cases. For example, Pair CodeT5 (2.90%) and Pair CodeT5+ (3.01%) slightly outperform CommentFinder (2.80%) and UniXCoder-IR (2.79%) in EM. BLEU scores show modest gains as well, with Pair CodeReviewer reaching 13.52% versus 12.41% and 12.80% for the IR baselines. While the margins are small, RAG-Reviewer offers the added benefit of generating comments conditioned on both the input code and retrieved exemplars, allowing it to leverage

TABLE IV
THE NUMBER OF CORRECTLY GENERATED LOW-FREQUENCY REVIEW TOKENS

Token Freq. ($\leq x$)	≤ 20	≤ 40	≤ 60	≤ 80	≤ 100
Vanilla Tufano T5	1102	1891	2450	2866	3238
Pair Tufano T5	1342	2282	3030	3508	3917
% Improvement	**21.78%**	**20.68%**	**23.67%**	**22.40%**	**20.97%**
Vanilla CodeReviewer	867	1525	2007	2405	2703
Pair CodeReviewer	1043	1869	2473	2936	3352
% Improvement	**20.30%**	**22.56%**	**23.22%**	**22.08%**	**24.01%**
Vanilla CodeT5	975	1770	2294	2711	3080
Pair CodeT5	1018	1835	2429	2876	3247
% Improvement	**4.41%**	**3.67%**	**5.88%**	**6.09%**	**5.42%**
Vanilla CodeT5+	1142	1956	2557	3089	3489
Pair CodeT5+	1229	2185	2909	3426	3856
% Improvement	**7.62%**	**11.71%**	**13.77%**	**10.91%**	**10.52%**
Vanilla AUGER	807	1583	2232	2708	3073
Pair AUGER	879	1751	2509	3043	3472
% Improvement	**8.92%**	**10.61%**	**12.41%**	**12.37%**	**12.98%**

semantic context that IR methods may overlook. Moreover, as shown in Table III, RAG-Reviewer's performance generally improves with stronger PLMs, suggesting that its effectiveness scales with the capacity of the underlying generator.

Answering RQ1

Pair retrieval consistently outperforms singleton retrieval in both EM and BLEU, and RAG-Reviewer achieves higher performance than generation-based and IR-based methods.

B. (RQ2) Does RAG-Reviewer mitigate the challenges faced by language models in generating low-frequency tokens?

Effect on Generating Low-frequency Tokens. A well-known challenge for language models in generation-based methods is their difficulty in generating low-frequency ground-truth tokens (LFGTs), due to a bias toward high-frequency patterns learned during training [17], [18]. In contrast, IR-based approaches can help expose such tokens by retrieving similar examples that contain them. While we have observed that RAG-Reviewer improves generation over standard baselines in RQ1, we further examine whether it specifically helps mitigate this limitation by evaluating its ability to generate LFGTs compared to generation-based methods. We define LFGTs as tokens that appear fewer than 100 times in the training review comment corpus following previous work [22], [30]. For each generated review comment, we count a token as correctly generated if it appears in both the output and the ground truth. We then group these correctly generated tokens by their frequency in the training set using thresholds of $\leq$20, $\leq$40, $\leq$60, $\leq$80, and $\leq$100. The improvement is measured as the relative increase over the generation-based baseline.

As shown in Table IV, RAG-Reviewer improves the generation of LFGTs across all evaluated models. Tufano T5 exhibits the most consistent relative gains, ranging from 20.68% to 23.67%. And CodeReviewer shows similar improvements

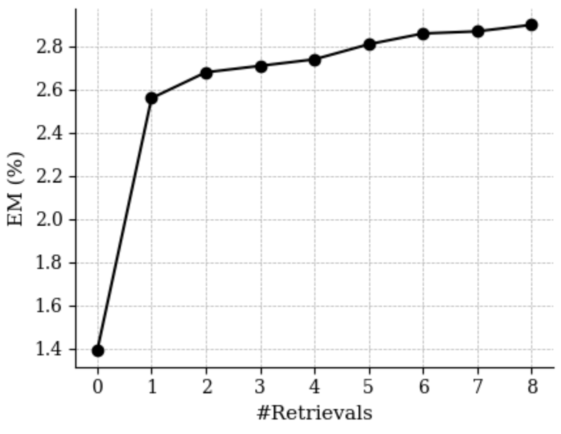


Fig. 6. Exact match performance of Pair CodeT5 with varying numbers of retrieved exemplars on Tufano et al. benchmark

(ranging from 20.30% to 24.01%), while AUGER demonstrates increases between 8.92% and 12.98%. CodeT5+ shows improvements of 7.62% to 13.77%, and CodeT5 achieves more modest gains between 3.67% and 6.09%. These results suggest that retrieval augmentation can help generation models recover rarer tokens by grounding generation in retrieved exemplars. While the level of improvement varies by model, the overall result indicates that RAG-Reviewer helps language model generate LFGTs more effectively.

Answering RQ2

RAG-Reviewer enhances the generation of low-frequency tokens across all models, alleviating innate limitation of generation-based methods.

C. *(RQ3) What is the impact of the number of retrieved exemplars on the performance of RAG-Reviewer?*

Impact of Retrieval Quantity on Generation Quality. While RQ1 showed that exemplar quality (Pair) is more important than quantity (Singleton), we further investigate whether increasing the number of high-quality exemplars still improves performance. To explore this, we evaluated the Pair CodeT5 model using retrieval sizes ranging from 0 (vanilla) to 8. This setup was chosen based on findings from RQ1: pair retrieval consistently outperformed singleton retrieval across all backbones, and CodeT5 showed mid-range performance among the RAG-Reviewer variants—making it a representative case for analysis.

As shown in Figure 6, exact match (EM) improves steadily from 1.39% with no retrieval (vanilla CodeT5) to 2.90% with eight exemplars. The biggest jump occurs when using just one exemplar (+1.17%), suggesting that even a small amount of retrieval helps the model generate more accurate comments. Additional exemplars continue to improve performance, but the gains become smaller. This is likely due to the model's input token limit, which restricts how much exemplar information can be included, thereby reducing the benefit of adding more examples.

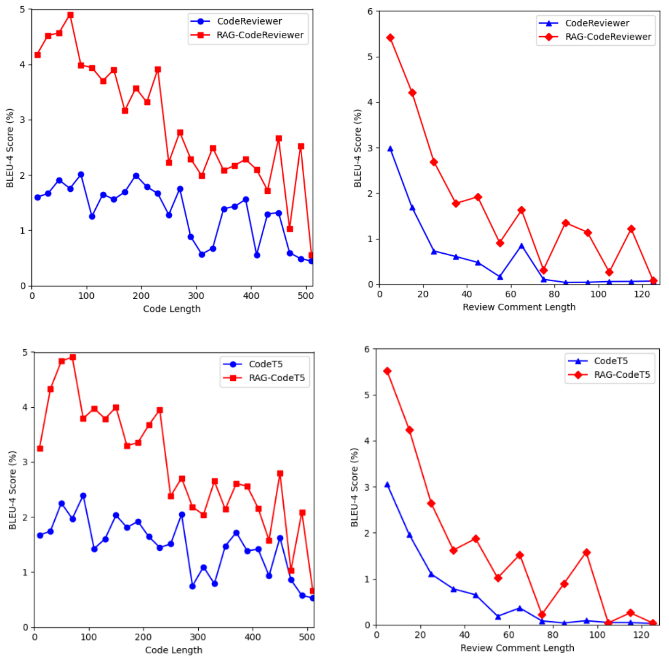


Fig. 7. BLEU-4 scores across different code and review comment lengths

Answering RQ3

Including more retrieved exemplars in the input consistently improves RAG-Reviewer's generation quality. However, performance gains diminish with higher counts due to input length constraints of the language model.

VI. Discussion

This discussion covers three parts: how RAG-Reviewer performs with different code and review comment lengths, findings from manual analysis, and directions for future work.

A. *Performance with Different Length*

To evaluate how RAG-Reviewer performs across varying input and output lengths, we compare BLEU-4 scores of Pair CodeReviewer and Pair CodeT5 with their vanilla counterparts. Test samples are bucketed by token length—20-token intervals for code and 10-token intervals for review comments—and average BLEU-4 is computed per bucket. As shown in Figure 7, both RAG models consistently outperform their vanilla versions across all length intervals. Although BLEU-4 scores decrease with longer sequences, the RAG models maintain relatively higher performance.

B. *Manual Analysis*

While RQ1 showed that RAG-Reviewer outperforms both IR-based and generation-based methods in terms of EM and BLUE, these metrics does not fully reflect the quality of

TABLE V
MANUAL ANALYSIS OF REVIEW COMMENTS ON 100 INPUT CODE FROM THE TUFANO ET AL. MANUAL ANALYSIS SAMPLES

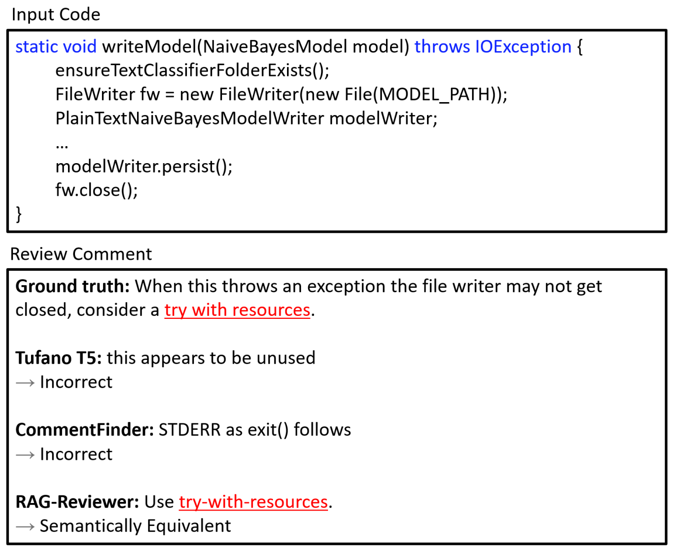
Category	RAG-Reviewer	CommentFinder	Tufano T5
Exact Match	2	4	0
Semantically Equivalent	39	30	36
Alternative Solution	9	9	10
Incorrect	50	57	54

TABLE VI
RETRIEVED EXEMPLARS USED FOR INPUT AUGMENTATION IN FIGURE 8 (KEY LFGT 'TRY-WITH-RESOURCES' UNDERLINED)

Top-K	Retrieved Review Comment
1	STDERR as exit() follows
10	Should we use try-with-resources here to close it later? Test only so I think it may not matter much.
16	This should be part of the try-with-resources block.
20	The try-with-resources takes care of the closing, right?

generated review comments. To evaluate outputs beyond these metrics, we conducted a manual analysis of 100 review comments generated by the best-performing RAG-Reviewer (Pair CodeT5+), using the same 100 samples manually analyzed in Tufano et al. [9], following the human evaluation method of prior work [9], [14]. Each generated review comment was classified into one of four categories based on the labeling scheme introduced in Tufano et al. [9]: (a) *Exact Match* – the generated review comment is character-wise identical to the ground-truth comment; (b) *Semantically Equivalent* – the generated review comment differs in wording but conveys the same meaning as the ground truth's intention; (c) *Alternative Solution* – the generated review comment is neither identical nor semantically equivalent but still provides a meaningful and useful suggestion; (d) *Incorrect* – the generated review comment is irrelevant and unhelpful for the given input code.

Table V summarizes the manual classification results. While RAG-Reviewer made fewer exact matches than CommentFinder, it produced more semantically equivalent comments (39 vs. 30) and fewer incorrect ones (50 vs. 57), indicating stronger relevance even when not perfectly matching the ground truth. Compared to Tufano T5, RAG-Reviewer yielded slightly fewer alternative solutions (9 vs. 10) but more semantically equivalent comments (39 vs. 36) and fewer incorrect outputs (50 vs. 54). Figures 8 and 9 show concrete examples. In Figure 8, RAG-Reviewer generates a semantically correct suggestion regarding `try-with-resources`, unlike the unrelated outputs from Tufano T5 and CommentFinder. Notably, RAG-Reviewer successfully generates the key LFGT `try-with-resources` (279 occurrences in the training corpus), which neither Tufano T5 nor CommentFinder produced. Although the input code does not contain this token, RAG-Reviewer can infer it from retrieved exemplars containing relevant context. As shown in Table VI, several retrieved examples (e.g., top-10, 16, and 20) include `try-with-resources`, allowing the model to refer to low-frequency but semantically important tokens. This demonstrates how retrieval augmentation provides contextual cues that help the model generalize beyond the input.


Input Code

```
static void writeModel(NaiveBayesModel model) throws IOException {
      ensureTextClassifierFolderExists();
      FileWriter fw = new FileWriter(new File(MODEL_PATH));
      PlainTextNaiveBayesModelWriter modelWriter;
      ...
      modelWriter.persist();
      fw.close();
}
```

Review Comment

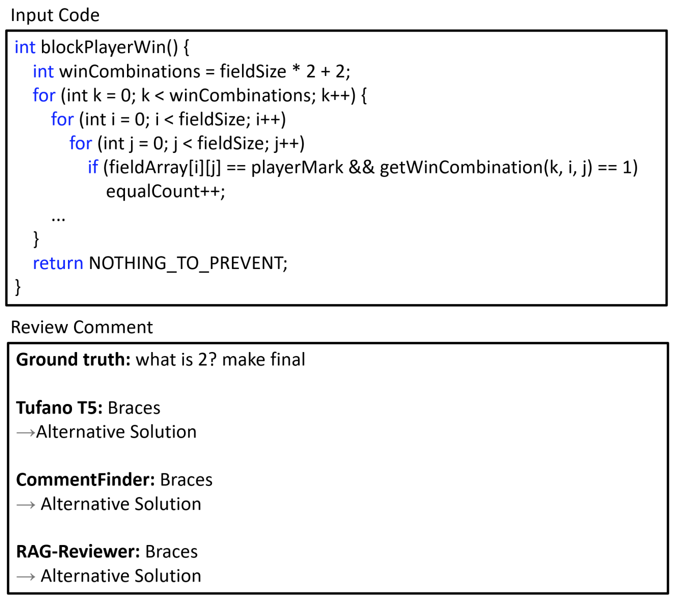
Ground truth: When this throws an exception the file writer may not get closed, consider a try with resources.

Tufano T5: this appears to be unused
→ Incorrect

CommentFinder: STDERR as exit() follows
→ Incorrect

RAG-Reviewer: Use try-with-resources.
→ Semantically Equivalent

Fig. 8. Example of a Semantically Equivalent review comment from the Tufano et al. manual analysis samples (Key LFGT 'try-with-resources' underlined)


Input Code

```
int blockPlayerWin() {
   int winCombinations = fieldSize * 2 + 2;
   for (int k = 0; k < winCombinations; k++) {
      for (int i = 0; i < fieldSize; i++)
         for (int j = 0; j < fieldSize; j++)
            if (fieldArray[i][j] == playerMark && getWinCombination(k, i, j) == 1)
               equalCount++;
      ...
   }
   return NOTHING_TO_PREVENT;
}
```

Review Comment

Ground truth: what is 2? make final

Tufano T5: Braces
→Alternative Solution

CommentFinder: Braces
→ Alternative Solution

RAG-Reviewer: Braces
→ Alternative Solution

Fig. 9. Example of a Alternative Solution review comment from the Tufano et al. manual analysis samples

C. Future Work

A key direction for future work is jointly training the retriever and generator. While this study fine-tuned only the generator, prior work [21] showed that training both components can improve retrieval quality and overall generation. Another direction is integrating RAG with large language models (LLMs). Although recent studies [12], [13] focused on efficient fine-tuning LLMs for RCG, combining them with RAG may further enhance their performance.

VII. Related Work

In this section, we review related work on Code Review Automation (CRA) from two main perspectives: generation-based and information retrieval-based (IR-based) approaches. We also discuss prior applications of retrieval-augmented generation (RAG) in software engineering.

A. Generation-based Code Review Automation

With the advancement of pretrained language models (PLMs) [43]–[47], there has been growing interest in applying them to various software engineering tasks [19], [29], [48], [49], including CRA. Tufano et al. [8], [9] were among the first to demonstrate the use of PLMs for CRA by curating a dataset of code–review comment pairs and framing the task as a sequence-to-sequence translation problem. Building on this, Li et al. [10] proposed specialized pretraining tasks designed to better capture the relationship between code changes (diffs) and corresponding review comments, further improving performance on CRA.

Recent works have continued to explore the use of PLMs on standard CRA benchmarks. For example, Zhou et al. [11] conducted a comparative study and found that CodeT5 [19] consistently outperformed other code-specific PLMs. Beyond PLMs, LLMs have also been adapted to CRA. Lu et al. [12] fine-tuned LLaMA [50] using Parameter-Efficient Fine-Tuning (PEFT) techniques and demonstrated competitive performance on the benchmark from Li et al. [10], though they reported that Tufano T5 [9] still outperformed LLaMA on the Tufano benchmark. Based on this finding, we opted to use PLMs rather than LLMs in our study, as they align better with the Tufano benchmark.

Expanding on Lu et al., Nashaat et al. [13] introduced a framework that incorporates organizational data with few-shot learning and reinforcement learning from human feedback. More recently, Li et al. [51] proposed CodeDoctor, a category-aware review comment generator that utilizes LLM-guided classification and a category-specific decoder. While CodeDoctor showed strong performance in multi-category review comment generation, it assumes predefined categories are available and constrains output accordingly. In contrast, our work focuses on general (non-categorical) review comment generation. This direction remains important for building fully automated and flexible CRA systems, as real-world code reviews often include issues that span or fall outside fixed categories.

B. Information Retrieval-based Code Review Automation

While generation-based CRA shows strong performance, it often incurs high computational cost during training and inference. IR-based CRA offers a more efficient alternative by bypassing language generation and instead reusing comments from similar examples in a retrieval corpus.

Hong et al. [14] proposed CommentFinder, an IR-based CRA system that uses sparse Bag-of-Words (BoW) vector representations to retrieve top-k lexically similar code snippets based on cosine similarity. The paired review comments from retrieved code snippets are returned as the predicted output. CommentFinder achieved a 32% improvement in exact match accuracy and was 49× faster than a generation-based baseline [9]. Building on this, Shuvo et al. [15] incorporated structural features such as class and library names to improve retrieval quality. More recently, Kartal et al. [16] demonstrated that dense representations from transformer-based models (e.g., UniXCoder [29]) outperform sparse representation (e.g., BoW) in CRA retrieval tasks.

C. Retrieval-Augmented Generation in Software Engineering

RAG combines the strengths of retrieval- and generation-based methods by incorporating retrieved exemplars into the input of a generative model. In software engineering, RAG has shown promise across several tasks, such as code summarization [21], [22], [24], [30], [52], code completion [23], and automated program repair [53].

Zhang et al. [22] were among the first to apply RAG to code summarization, using LSTM-based models. Parvez et al. [24] extended this idea by using transformer-based PLMs as both retriever and generator. Lu et al. [23] applied RAG to code completion by retrieving similar completed code examples to guide the model. Jin et al. [53] introduced RAG into LLM-based program repair by augmenting prompts with retrieved bug–fix pairs to improve repair quality.

VIII. Threats to Validity

Internal validity. We reimplemented Tufano T5 in PyTorch due to compatibility issues with its original TensorFlow 2.6.0 code. Pretrained weights were converted accordingly, and the training setup followed prior RCG work [10], [12]. While care was taken to align with the original setup, minor discrepancies may exist. Hyperparameters were selected based on prior work and GPU memory limits but may not be optimal.

External validity. Our evaluation is based solely on the Java dataset from Tufano et al. [9]. To improve generalizability, future work should evaluate across more languages and diverse software projects.

IX. Conclusion

In this paper, we presented **RAG-Reviewer**, a **RAG**-based code **Review** comment gen**er**ation framework. RAG-Reviewer conditions pretrained language models on both input code and retrieved code–review exemplars to improve generation quality. We evaluated the approach on the Tufano et al. benchmark and observed consistent gains over generation-based and IR-based baselines, particularly in generating low-frequency tokens and handling varying input lengths. Our analysis further reveals that using fewer but more informative exemplars—including relevant code—outperforms comment-only augmentation, and that increasing the number of exemplars within the same retrieval strategy leads to additional improvements. These results demonstrate the effectiveness of retrieval augmentation in enhancing code review automation.

REFERENCES

[1] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, pp. 2146–2189, 2016.

[2] E. Fregnan, F. Petrulio, L. Di Geronimo, and A. Bacchelli, "What happens in my code reviews? an investigation on automatically classifying review changes," *Empirical Software Engineering*, vol. 27, no. 4, p. 89, 2022.

[3] Y. Huang, X. Liang, Z. Chen, N. Jia, X. Luo, X. Chen, Z. Zheng, and X. Zhou, "Reviewing rounds prediction for code patches," *Empirical Software Engineering*, vol. 27, pp. 1–40, 2022.

[4] A. Bosu and J. C. Carver, "Impact of peer code review on peer impression formation: A survey," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 133–142.

[5] Y. Hong, C. K. Tantithamthavorn, and P. P. Thongtanunam, "Where should i look at? recommending lines that reviewers should pay attention to," in *2022 IEEE international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2022, pp. 1034–1045.

[6] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, 2013, pp. 202–212.

[7] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[8] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 163–174.

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

[10] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu *et al.*, "Codereviewer: Pre-training for automating code review activities," *arXiv preprint arXiv:2203.09095*, 2022.

[11] X. Zhou, K. Kim, B. Xu, D. Han, J. He, and D. Lo, "Generation-based code review automation: How far are we?" in *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE, 2023, pp. 215–226.

[12] J. Lu, L. Yu, X. Li, L. Yang, and C. Zuo, "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, 2023, pp. 647–658.

[13] M. Nashaat and J. Miller, "Towards efficient fine-tuning of language models with organizational data for automated software review," *IEEE Transactions on Software Engineering*, 2024.

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

[15] O. Shuvo, P. Mahbub, and M. M. Rahman, "Recommending code reviews leveraging code changes with structured information retrieval," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2023, pp. 194–206.

[16] Y. Kartal, E. K. Akdeniz, and K. Özkan, "Automating modern code review processes with code similarity measurement," *Information and Software Technology*, vol. 173, p. 107490, 2024.

[17] P. Arthur, G. Neubig, and S. Nakamura, "Incorporating discrete translation lexicons into neural machine translation," *arXiv preprint arXiv:1606.02006*, 2016.

[18] J. Zhang, M. Utiyama, E. Sumita, G. Neubig, and S. Nakamura, "Guiding neural machine translation with retrieved translation pieces," *arXiv preprint arXiv:1804.02559*, 2018.

[19] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[20] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.

[21] H. Lu and Z. Liu, "Improving retrieval-augmented code comment generation by retrieving for generation," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2024, pp. 350–362.

[22] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1385–1397.

[23] S. Lu, N. Duan, H. Han, D. Guo, S.-w. Hwang, and A. Svyatkovskiy, "Reacc: A retrieval-augmented code completion framework," *arXiv preprint arXiv:2203.07722*, 2022.

[24] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Retrieval augmented code generation and summarization," *arXiv preprint arXiv:2108.11601*, 2021.

[25] J. Shin, R. Aleithan, H. Hemmati, and S. Wang, "Retrieval-augmented test generation: How far are we?" *arXiv preprint arXiv:2409.12682*, 2024.

[26] X. Han, A. Tahir, P. Liang, S. Counsell, and Y. Luo, "Understanding code smell detection via code review: A study of the openstack community," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 323–334.

[27] A. Alami, M. L. Cohn, and A. Wąsowski, "Why does code review work for open source software communities?" in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1073–1083.

[28] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at google," in *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, 2018, pp. 181–190.

[29] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[30] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: exemplar-based neural comment generation," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 349–360.

[31] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[32] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of machine learning research*, vol. 21, no. 140, pp. 1–67, 2020.

[33] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[34] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint arXiv:2305.07922*, 2023.

[35] L. Li, L. Yang, H. Jiang, J. Yan, T. Luo, Z. Hua, G. Liang, and C. Zuo, "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*, 2022, pp. 1009–1021.

[36] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Tufano t5 replication package," https://github.com/RosaliaTufano/code_review_automation, 2021, accessed: 2025-05-24.

[37] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "Codet5-base replication package," https://huggingface.co/Salesforce/codet5-base, 2021, accessed: 2025-05-24.

[38] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, "Codet5p-220m replication package," https://huggingface.co/Salesforce/codet5p-220m, 2023, accessed: 2025-05-24.

[39] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu *et al.*, "Codereviewer replication package," https://huggingface.co/microsoft/codereviewer, 2022, accessed: 2025-05-24.

[40] L. Li, L. Yang, H. Jiang, J. Yan, T. Luo, Z. Hua, G. Liang, and C. Zuo, "Auger replication package," https://gitlab.com/ai-for-se-public-data/auger-fse-2022, 2022, accessed: 2025-05-24.

[41] Y. Hong, C. Tantithamthavorn, P. Thongtanunam, and A. Aleti, "Commentfinder replication package," https://github.com/awsm-research/CommentFinder, 2022, accessed: 2025-05-24.

[42] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder replication package," https://huggingface.co/microsoft/unixcoder-base-nine, 2021, accessed: 2025-05-24.

[43] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

[44] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[45] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[46] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.

[47] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[48] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[49] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[50] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[51] Y. Li, Y. Wu, Z. Wang, L. Huang, J. Wang, J. Li, and M. Huang, "Codedoctor: multi-category code review comment generation," *Automated Software Engineering*, vol. 32, no. 1, p. 25, 2025.

[52] J. A. Li, Y. Li, G. Li, X. Hu, X. Xia, and Z. Jin, "Editsum: A retrieve-and-edit framework for source code summarization," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 155–166.

[53] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1646–1656.