Automatic Comment Generation via Multi-Pass Deliberation

Fangwen Mu*†
fangwen2020@iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

Xiao Chen*†
chenxiao2021@iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

Lin Shi*†‡
shilin@iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

Song Wang
wangsong@eecs.yorku.ca
York University, Lassonde School of Engineering
Toronto, Canada

Qing Wang*†‡§
wq@iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

ABSTRACT

Deliberation is a common and natural behavior in human daily life. For example, when writing papers or articles, we usually first write drafts, and then iteratively polish them until satisfied. In light of such a human cognitive process, we propose DECOM, which is a multi-pass deliberation framework for automatic comment generation. DECOM consists of multiple *Deliberation Models* and one *Evaluation Model*. Given a code snippet, we first extract keywords from the code and retrieve a similar code fragment from a pre-defined corpus. Then, we treat the comment of the retrieved code as the initial draft and input it with the code and keywords into DECOM to start the iterative deliberation process. At each deliberation, the deliberation model polishes the draft and generates a new comment. The evaluation model measures the quality of the newly generated comment to determine whether to end the iterative process or not. When the iterative process is terminated, the best-generated comment will be selected as the target comment. Our approach is evaluated on two real-world datasets in Java (87K) and Python (108K), and experiment results show that our approach outperforms the state-of-the-art baselines. A human evaluation study also confirms the comments generated by DECOM tend to be more readable, informative, and useful.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**.

KEYWORDS

Code Summarization, Information Retrieval, Deep Neural Network

*Also With Laboratory for Internet Software Technologies, Institute of Software, CAS
†Also With University of Chinese Academy of Sciences
‡Corresponding author
§Also With State Key Laboratory of Computer Sciences, Institute of Software, CAS

This work is licensed under a Creative Commons Attribution International 4.0 License.

ASE '22, October 10–14, 2022, Rochester, MI, USA

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3556917

ACM Reference Format:
Fangwen Mu, Xiao Chen, Lin Shi, Song Wang, and Qing Wang. 2022. Automatic Comment Generation via Multi-Pass Deliberation. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556917

1 INTRODUCTION

Code comment generation concerns the production of a concise and fluent description of source code that facilitates software development and maintenance by enabling developers to comprehend, ideate, and document code effectively. Most of the existing approaches treat comment generation as a machine translation task and adopt a one-pass encoder-decoder process, i.e., first encode the input code into a sequence of semantic features, then decode the features to a natural language comment [24, 26, 30, 57]. Although the encoder-decoder framework has achieved remarkable performance on the comment generation task, it still suffers from two major limitations. The first one is that, such models adopt a regular one-pass decoding process which sequentially generates comments word by word. They directly use the generated comment as the final output, which results in their inability to correct the mispredicted words. The words mistakenly predicted in the early steps may lead to error accumulation under the constraint of the language model.

Taking the auto-generated comments in Figure 1 as an example, the one-pass model Re^{2}com [52] incorrectly predicts the sixth word "probability" as "bytes", which leads to the model to keep exploring the words related to "bytes" when predicting the consequential words. As a result, the related words "written to this stream" are mistakenly generated, which results in a typical example of error accumulation. The second limitation is that they generate comments sequentially. Thus, such sequential one-pass models cannot leverage the global information of the generated comment to further polish its local content. As the example shown in Figure 1, the one-pass model EditSum [30] generates two consecutive prepositional phrases after the word "copy". Although either of them is reasonable in their local contexts ("of the given table" and "of the table"), putting them together degrades the comment fluency and makes the developers hard to understand.

To alleviate these challenges, we introduce the *Deliberation* mechanism [56] in the comment generation task, aiming to further enhance the performance. Deliberation is a common and natural

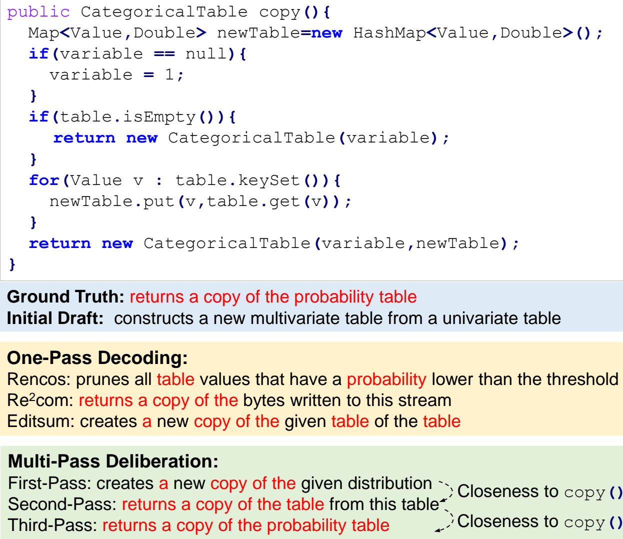

Figure 1: A motivation example of multi-pass deliberation.

behavior in human daily life. When writing papers or articles, we usually first write drafts, and then iteratively polish them until satisfied. Figure 1 illustrates an example of applying multi-pass deliberation on comment generation. Based on the initial draft "constructs a new multivariate table from a univariate table", the first-pass deliberation will generate the comment "creates a new copy of the given distribution", and refine it in the second and the third pass guided by the closeness to the similarity with the give code snippet. In the end, we could obtain the most satisfying comment "returns a copy of the probability table".

In light of such a human cognitive process, we propose a novel multi-pass deliberation framework for automatic comment generation, named DECOM, which contains multiple *Deliberation Models* and one *Evaluation Model*. Given a code snippet, initially, we retrieve the most similar code from a pre-defined corpus and treat its comment as the initial draft. We also extract the identifier names from the input code as keywords, since these user-defined words usually contain more semantic information that users want to express [10, 43, 47]. Then, we input the code, the keywords, and the initial draft into DECOM to start the iterative deliberation process. At each deliberation, the deliberation model polishes the draft and generates a new comment. The evaluation model calculates the quality score of the newly generated comment. This multi-pass process terminates when (1) the quality score of the new comments is no longer higher than the previous ones, or (2) the maximum number of deliberation is reached. To evaluate our approach, we conduct experiments on two real-world datasets in Java (87K) and Python (108K), and the results show that our approach outperforms the state-of-the-art baselines by 8.3%, 6.0%, 13.3%, and 10.5% with respect to BLEU-4, ROUGE-L, METEOR, and CIDEr on Java dataset. On Python dataset, DECOM improves the performance on BLEU-4, ROUGE-L, METEOR, and CIDEr by 5.8%, 3.8%, 6.6%, and 6.3%, respectively. We also conduct a human evaluation to assess the generated comments on three aspects: naturalness, informativeness, and usefulness, showing that DECOM can generate useful and relevant comments.

Our main contributions are outlined as follows:

- **Technique**: a multi-pass deliberation framework for comment generation, named DECOM, which is inspired by the human cognitive process, and can effectively generate comments in an iterative way. To the best of our knowledge, this is the first work that employs multi-pass deliberation to enhance the performance of comment generation.
- **Evaluation**: an experimental evaluation of the performance of DECOM against state-of-the-art baselines, which shows that DECOM outperforms all baselines, together with a human evaluation, which further confirms the readability, informativeness, and usefulness of DECOM.
- **Data**: publicly accessible dataset and source code [4] to facilitate the replication of our study and its application in extensive contexts.

In the rest of this paper, Section 2 elaborates the approach. Section 3 presents the experimental setup. Section 4 demonstrates the results and analysis. Section 5 describes the human evaluation. Section 6 discusses indications and threats to validity. Section 7 introduces the related work. Section 8 concludes our work.

2 APPROACH

In this section, we present our DECOM, a multi-pass deliberation framework that performs an iterative polishing process to refine the draft to a better comment. Figure 2 illustrates an overview of DECOM, which consists of three main stages: (1) **Data initialization**, for extracting the keywords from the input code and retrieving the similar code-comment pair from the retrieval corpus; (2) **Model training**, for leveraging a two-step training strategy to optimize DECOM; and (3) **Model prediction**, for generating the target comment of the new source code. Below, we provide details for each stage in DECOM.

2.1 Data Initialization

Given a code snippet x, this stage aims to extract the keywords t from x, and retrieve the initial draft z^0 from the retrieval corpus.

Extract keywords from code. A code snippet contains many different types of tokens, such as reserve words (*if*, *for*), identifier names (*set_value*, *SortList*), and operators (+, ∗). Among them, identifier names defined by users usually contain more semantic information that users want to express [10, 43, 47]. For example, a method's name is a typical identifier name, which is used to describe the overall functionality of the code and can be considered as a shorter version of its code comment. Thus, to enable the model to attend more on the identifier names and capture semantic information from them, we extract these words from code. First, we use the *javalang* [3] and *tokenize* [5] libraries to extract identifier names from the Java and Python code snippets, respectively. Then, we further split the extracted identifier names into sub-tokens by *CamelCase* or *snake_case* to obtain the smaller semantic units and reduce data sparsity. These sub-tokens are treated as the keywords of the code.

Retrieve the initial draft. To obtain the initial draft, following the previous studies [52], we use the lexical similarity-based retrieval method to identify the top similar code-comment pair for the given code x. Specifically, we first take the training set of the benchmark dataset as the retrieval corpus. Then, for each code in

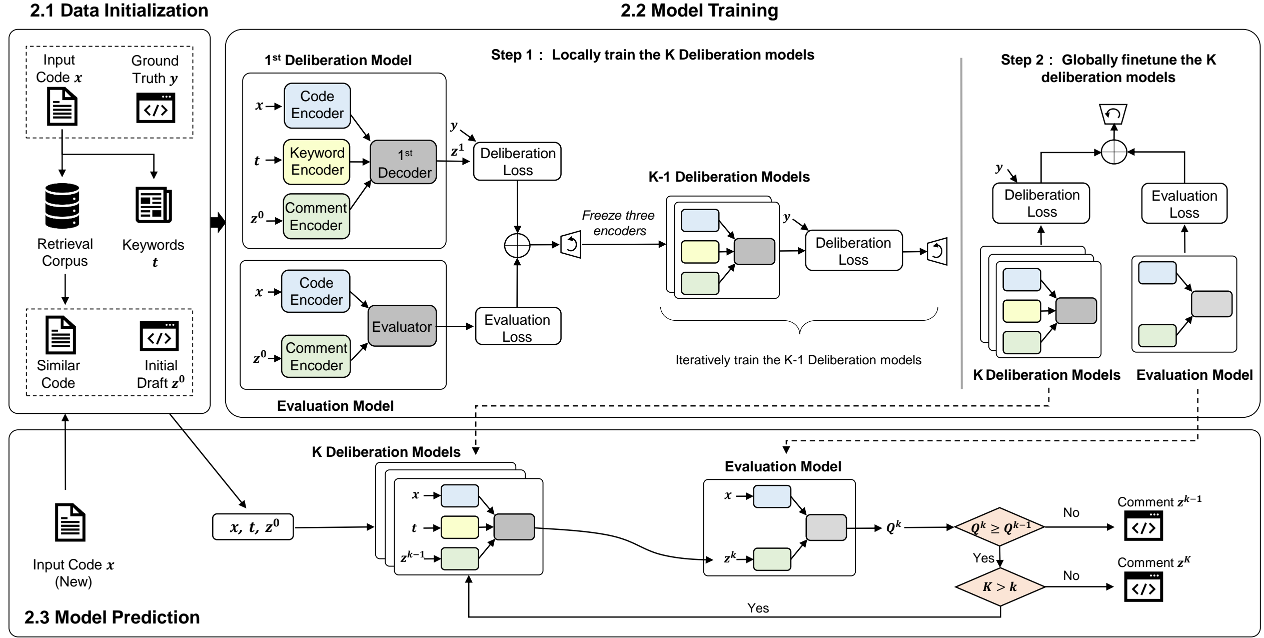

Figure 2: The overall architecture of DECOM

the retrieval corpus, we adopt the BM25 [40] metric to calculate the similarity between it and the given code x. The BM25 is a bag-of-words retrieval metric to measure the relevance of documents to a given search query in IR and is also widely used in code clone detection and code search tasks [27, 35, 41]. Finally, we extract the code with the highest similarity score as the retrieved result, and use the comment of the code as the initial draft z^0. Since the size of our training sets is quite large, we leverage the open-source search engine Lucene [1] to speed up the retrieval process. We follow the settings of Lucene from Re^{2}Com [52] to run our experiments.

2.2 Model Training

DECOM contains K deliberation models and one evaluation model, where K is the maximum number of deliberation. To reduce computation cost and facilitate the sharing of information between models, all K deliberation models share three encoders with others and share the code encoder and comment encoder with the evaluation model. Each deliberation model has its own decoder, which can avoid these models generating highly similar comments. We employ a two-step training strategy to train DECOM as shown in Figure 2. In the first step, we locally train the K deliberation models: we first jointly train the first deliberation model and the evaluation model. Then we freeze the shared encoders and train the other deliberation models one by one. In the second step, we fine-tune DECOM by jointly optimizing all trained models.

2.2.1 *Deliberation Model.*

Each deliberation model consists of three different encoders (i.e. code encoder, keyword encoder, and comment encoder) and a decoder. The details of them are illustrated in the following.

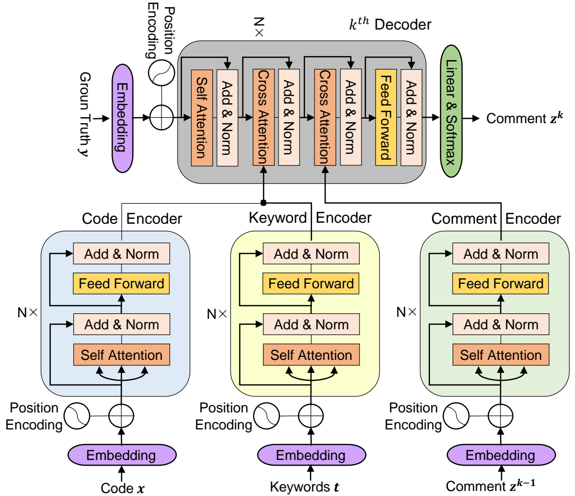

Figure 3: The detailed structure of the Deliberation model.

Encoders. The code encoder, keyword encoder, and comment encoder aim to encode the source code x, keywords t, and the previous comment z^{k-1} as vectors, which enables the deliberation model to obtain the semantics from both source-side (code and keywords) and target-side (the past comment). We construct the three encoders by following the structure of the vanilla Transformer Encoder [48]. As shown in Figure 3, each encoder is composed of a stack of N identical Transformer blocks. Each block contains two sub-layers: The first sub-layer is a multi-head self-attention layer (MHAtt), which employs multiple attention heads to capture the information from different representation sub-spaces at different positions. The second sub-layer is a two-layer Feed-Forward Network (FFN) with

a ReLU activation function in between. The residual connection is employed around the two sublayers, followed by layer normalization (LayerNorm) [7]. Since the three encoders have the same structure, we only introduce the code encoder for simplicity.

Given a code snippet $x = [x_1, x_2, ..., x_{l(x)}]$, where $l(x)$ is the number of words in the code. The code encoder first embeds each word of the code into a d dimensional word vector:

$$\overrightarrow{x_i} = {W_e}^T \cdot x_i + PE_i \tag{1}$$

where W_e is a trainable embedding matrix, and PE_i is the position encoding of the i-th word. Following previous study [48], we use the *sine* and *cosine* function of different frequencies to compute the position encoding:

$$PE_{i,2j} = \sin\left(j/10000^{2j/d}\right) \tag{2}$$

$$PE_{i,2j+1} = \cos\left(j/10000^{2j/d}\right) \tag{3}$$

where i is the position of the word and j denotes the j-th dimension of the embedding vector.

Then, the code encoder inputs the sequence of word embeddings into N identical Transformer blocks to calculate the hidden states of the code. For the ith block of the code encoder, suppose that the input is H_{i-1}, the output H_i is calculated as follows:

$$H_{i,1} = \text{LayerNorm}\Big(H_{i-1} + \text{MHAtt}(H_{i-1}, H_{i-1}, H_{i-1})\Big) \tag{4}$$

$$H_i = \text{LayerNorm}\Big(H_{i,1} + \text{FFN}(H_{i,1})\Big) \tag{5}$$

where $H_{i,1}$ is the hidden states of the first sub-layer. Initially, the word embedding vectors $[\overrightarrow{x_1}, \overrightarrow{x_2}, ..., \overrightarrow{x_{l(x)}}]$ are fed into the first block, and the Nth block outputs the final hidden states of the input code $H = [h_1, h_2, ..., h_{l(x)}]$. Similarly, DECOM can encode the keywords t and the past comment z^{k-1} into hidden states P and R^{k-1}, respectively.

There are two points worth noting: (1) In the first-pass deliberation, DECOM takes the comment of the retrieved code as the initial draft z^0, for each turn after this, DECOM uses the comment generated in the previous turn as the draft. (2) source code x and keywords t do not change in the iterative deliberation process, so to save computational resources and time, we compute their hidden states H and P only once, and reuse them in subsequent iterations.

Decoder. The decoder aims to improve the quality of the previously generated comment z^{k-1} by jointly leveraging its context and the semantics of the source code x and the keywords t. As shown in Figure 3, the decoder is also composed of a stack of N identical decoder blocks, and each block consists of four sub-layers. In addition to the first and the last sub-layers introduced in the part of Encoders in section 2.2.1, the decoder block inserts two multi-head cross attention sub-layers in between, which are used to capture the information from the outputs of the three encoders.

In the kth pass deliberation ($k \geq 1$), given the hidden states H, P, R^{k-1}. The ith block of the decoder first gets the hidden states of the first sub-layer $S_{i,1}$ using Eq. (4). Then, in the second sub-layer, the block separately performs multi-head attention over the hidden states of the source code H and the keywords P:

$$a_i = \text{MHAtt}(S_{i,1}, H, H) \tag{6}$$

$$b_i = \text{MHAtt}(S_{i,1}, P, P) \tag{7}$$

In order to effectively leverage the information from source-side, we utilize the gate mechanism [21] to adaptively incorporate the a_i containing source code features and the b_i containing keywords features:

$$\beta = \text{Sigmoid}(W_{gate}^T[a_i \ ; \ b_i]) \tag{8}$$

$$S_{i,2} = \beta \cdot a_i + (1-\beta) \cdot b_i \tag{9}$$

where β is the degree of integration between source code and keywords, A larger value of the β (ranges from 0 to 1) indicates that the model should pay more attention to the information in the source code. W_{gate} is a trainable parameter matrix, operation $[;]$ is concatenation, and $S_{i,2}$ is the hidden states of the second sub-layer. In the third sub-layer, the block obtains the $S_{i,3}$ by performing multi-head attention over the hidden states of the previous comment R^{k-1}:

$$S_{i,3} = \text{LayerNorm}\Big(S_{i,2} + \text{MHAtt}(S_{i,2}, R^{k-1}, R^{k-1})\Big) \tag{10}$$

Based on this equation, the decoder can capture the important clues from the global information of the past comment for further refinement. Then, according to Eq. (5), the ith block uses the $S_{i,3}$ to compute the output of the last sub-layer S_i. After the calculation of N decoder blocks, the decoder gets the hidden states of the last decoder block S. For the j-th decoding step, the probability of jth token z_j^k can be calculated by projecting the jth state s_j in S via a linear layer followed by a Softmax function.

$$p(z_j^k | z_1^k, \ z_2^k, \ ..., \ z_{j-1}^k) = \text{Softmax}(W_o^T \cdot s_j + b_o) \tag{11}$$

where W_o is the parameter matrix and b_o is the bias. Ultimately, we use the Argmax function to generate the new comment z^k.

$$z^k = \text{Argmax}([p(z_1^k) \ ; \ p(z_2^k) \ ; \ \cdots \ ; \ p(z_{l(k)}^k)]) \tag{12}$$

where the $l(k)$ is the length of the kth generated comment.

2.2.2 Evaluation Model. The evaluation model aims to estimate the quality of the generated comments and calculate their quality scores. As shown in Figure 4, the evaluation model contains a shared code encoder, a shared comment encoder, and an evaluator.

Given the new comment z^k generated by the kth deliberation model, the comment encoder encodes the z^k into hidden states R^k. To obtain the representation of the comment, the evaluator first uses Mean Pooling to average the hidden states $R^k = [r_0^k, r_1^k, ..., r_{l(k)}^k]$ to get the aggregated features v_{mean}^k. Then, it utilizes a two-layer feed-forward network (FFN) to map the features into the comment representation v^k.

$$v_{mean}^k = \text{MeanPooling}([r_0^k \ ; \ r_1^k \ ; \ \cdots \ ; \ r_{l(k)}^k]) \tag{13}$$

$$v^k = \text{FFN}(v_{mean}^k) = \text{ReLU}(v_{mean}^k \cdot W_1 + b_1) \cdot W_2 + b_2 \tag{14}$$

Similarly, the evaluator can obtain the code representation v^x using the Eq. (13) and (14) . Then, we use the cosine similarity metric to calculate the similarity score Q^k between v^k and v^x. A higher

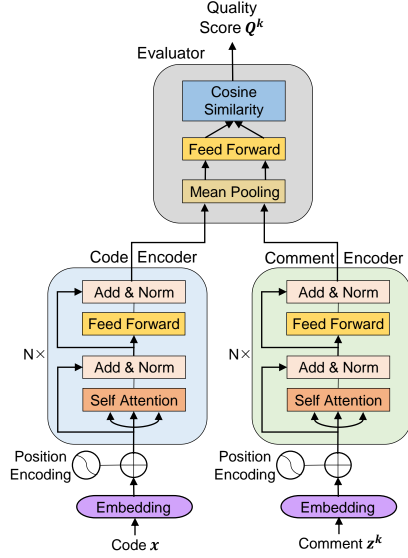

Figure 4: The detailed structure of the Evaluation model.

similarity score indicates that the comment z^k is more semantically similar to the source code x.

$$Q^k = Cos(v^x,\ v^k) = \frac{v^{x\,T} \cdot v^k}{\| v^x \| \cdot \| v^k \|} \tag{15}$$

2.2.3 Two-Step Training. In this section, we describe the training process and strategies for DECOM. We denote the parameters of the kth deliberation model as θ_d^k and the parameters of the evaluation model as θ_e.

Deliberation loss and Evaluation loss. Given the source code x, the ground truth y, the keywords t, and the previous comment z^{k-1}, the kth deliberation model can be optimized by maximizing the probability of $p(y|x, t, z^{k-1})$. The loss function is calculated as:

$$\mathcal{L}_{delib}(\theta_d^k) = \sum_{1 \le i \le l(y)} -\log p\left(y_i | y_{<i}, x, t, z^{k-1}\right) \tag{16}$$

where $l(y)$ is the length of ground truth y. For the evaluation model, we use the Circle Loss function [46] to optimize its parameters θ_e:

$$\mathcal{L}_{eval}(\theta_e) = log\left(1 + e^{\lambda(Cos(v^x, v^k) - Cos(v^x, v^y))}\right) \tag{17}$$

where $Cos()$ denotes the cosine similarity score, v^x, v^k and v^y are the representation vectors of the source code, the kth generated comment, and the ground truth, respectively.

Two-step Training Strategy. In theory, we can train the framework from random initialization by jointly optimizing all components. However, we find that training the multi-pass model directly from scratch is unstable in practice, which is mainly because of the cold start [44] problem. To mitigate this problem, we use a two-step training strategy as shown in Figure 2.

Step 1: Locally train the K Deliberation models. We first jointly train the first deliberation model and the evaluation model by minimizing the following loss function:

$$\mathcal{L}(\theta_d^1, \theta_e) = \mathcal{L}_{delib}(\theta_d^1) + \alpha_e \mathcal{L}_{eval}(\theta_e) \tag{18}$$

where the α_e is a hyperparameter, which is set to be 0.1 in our experiments to control the weight of the evaluation loss. Then we freeze the three shared encoders, and iteratively train the subsequent deliberation models using the Eq. (16) until the last deliberation model is trained.

Step 2: Globally train the K Deliberation models. One of the drawbacks of the first-step training is that the deliberation models is optimized independently and the model components cannot share the information. To address this, we further fine-tune DECOM by jointly training all K deliberation models and the evaluation model:

$$\mathcal{L}(\theta_d^1, ..., \theta_d^K, \theta_e) = \mathcal{L}_{delib}(\theta_d^1) + ... + \mathcal{L}_{delib}(\theta_d^K) + \alpha_e \mathcal{L}_{eval}(\theta_e) \tag{19}$$

Note that in this step, all parameters are unfrozen and are updated at the same time.

2.3 Model Prediction

The prediction stage aims to generate a concise and useful comment for a given code snippet. As shown in Figure 2, given a new code snippet x, we first perform data initialization introduced earlier to obtain the keywords t and the initial draft z^0. Then, we input them into DECOM to generate the target comment automatically. The comment generation process involves multiple deliberation processes. During the kth deliberation, the kth deliberation model polishes the previously generated comment z^{k-1} and generates a new comment z^k. The evaluation model estimates the quality of the new comment z^k by calculating the cosine similarity between this and the source code x. The deliberation process is performed iteratively unless either of the following two conditions is satisfied: (1) the quality score of the new comment is no longer higher than the previous ones; (2) a certain number of deliberation $K > 0$ is reached. In the former case, we adopt the previous comment as the target comment. In the latter case, the last generated comment is accepted.

3 EXPERIMENTAL SETUP

3.1 Dataset

Since most of the related studies [6, 12, 15, 57, 58] for comment generation tasks are evaluated on JCSD [25] and PCSD [9] benchmark datasets, in this study, we also select these two datasets in our experiments. JCSD has 87,136 code-comment pairs collected from more than 9K Java Github repositories created from 2015 to 2016 with at least 20 stars. It first extracted Java methods and Javadocs, and treated the first sentence of the Javadoc as the ground-truth comment of the corresponding code. PCSD contains 108,726 code-comment pairs collected from open source repositories on GitHub. It used docstrings (i.e., the string literals that appear right after the definition of functions) as comments for Python functions.

For the sake of fairness, we preprocess the JCSD and PCSD strictly following Rencos [57]. Specifically, we first split datasets into a training set, validation set, and test set in a consistent proportion of 8 : 1 : 1 for the Java dataset and 6 : 2 : 2 for the Python dataset. We use the *javalang* [3] and *tokenize* [5] libraries to tokenize the code snippet for JCSD and PCSD, respectively. We further split code tokens of the form *CamelCase* and *snake_case* to respective subtokens. For JCSD, we remove the exactly duplicated code-comment

pairs in the test set. The specific statistics of the two preprocessed datasets are shown in Table 1.

Table 1: Statistic of Datasets

Dataset	JCSD	PCSD
Train	69,708	65,236
Validation	8,714	21,745
Test	6,489	21,745
Unique tokens in code	230,336	481,756
Unique tokens in comment	35,535	37,111
Avg. tokens in code	99.9	133.1
Avg. tokens in comment	17.1	9.9
Max. token in code	4,842	157,116
Max. token in comment	670	333

3.2 Evaluation Metrics

We evaluate the performance of different approaches using common metrics including BLEU [38], ROUGE-L [33], METEOR [8], and CIDEr [49]. **BLEU** measures the n-gram precision by computing the overlap ratios of n-grams and applying brevity penalty on short translation hypotheses. BLEU-1/2/3/4 corresponds to the scores of unigram, 2-grams, 3-grams, and 4-grams, respectively. **ROUGE-L** is defined as the length of the longest common subsequence between generated sentence and reference, and based on recall scores. **METEOR** is based on the harmonic mean of unigram precision and recall, with recall weighted higher than precision. **CIDEr** considers the frequency of n-grams in the reference sentences by computing the TF-IDF weighting for each n-gram. $CIDEr_n$ score for n-gram is computed using the average cosine similarity between the candidate sentence and the reference.

3.3 Implementation Details

Following previous studies [57], we set the length limits (in terms of #words) of code and comment (i.e., 300 and 30 for JCSD, 100 and 50 for PCSD). To save the computing resource, we limit the maximum vocabulary size of source code and comment to 50K for both datasets. The out-of-vocabulary words are replaced by 'UNK'. The word embedding size of both code and comment is set to 512. We set the dimensions of hidden states to 512, the number of heads to 8, and the number of blocks to 6, respectively. The maximum iteration number K is set to be 3. We set the mini-batch size to 32 and train our approach using the Adam [28] optimizer. In the first-step training, we set the learning rate to 1e-4, and for the second-step training, we use a smaller learning rate (1e-5) to fine-tune DECOM. To avoid the over-fitting problem, we apply dropout [20] with 0.2. The maximum number of epochs is set to 100 for each step of training. We also use the strategy of early stopping, when the validation performance does not improve for 20 consecutive epochs, the training process will be stopped. To reduce training time, we use the greedy search to generate comments at the training stage. During the prediction stage, we use the beam search [53] and set the beam size to 5 for choosing the best result. Our approach is implemented based on the Pytorch [2] framework. The experimental environment is a desktop computer equipped with an NVIDIA GeForce RTX 3060 GPU, intel core i5 CPU, and 12GB RAM, running on Ubuntu OS.

4 RESULTS

We address the following three research questions to evaluate the performance of DECOM:

RQ1 : How does the DECOM perform compared to the state-of-the-art comment generation baselines?

RQ2: How does each individual component in DECOM contribute to the overall performance?

RQ3: What's the performance of DECOM on the data with different code or comment length?

4.1 RQ1: Comparison with Baselines

4.1.1 Baselines. We compare our approach with three categories of existing work on the comment generation task. We exactly adopt the hyperparameter settings reported in the original paper for all baselines. For a fair comparison, we use the same maximum code and comment length for all approaches, and evaluate their performance using the same training/testing datasets.

- IR-based baselines. **LSI [13]** is an IR technique to analyze the latent meaning or concepts of documents. The similarity between the code and the comment is computed based on the LSI-reduced vectors and cosine distance, and we set the vector dimension to be 500. **VSM [42]** is also a commonly used IR technique in comment generation tasks. For a given code snippet, we represent the code as a vector using TF-IDF, and extract the summary of the most similar code based on cosine similarity. **NNGen [34]** is a nearest-neighbors approach for generating commit messages. It first embeds code into vectors based on the bag of words and the term frequency. Then, it retrieves the nearest neighbors of the code. Finally, it outputs the message of the code with the highest BLEU score.
- NMT-based approaches. **CODE-NN [26]** is the first learning-based model for comment generation. It maps the source code sequence into word embeddings, then uses the LSTM and the attention mechanism to generate comments. **TL-CodeSum [25]** is a multi-encoder neural model that encodes API sequences along with code token sequences and generates summaries from source code with transferred API knowledge. **Hybrid-DRL [50]** incorporates ASTs and sequential content of code snippets into a deep reinforcement learning framework.
- Hybrid approaches. **Rencos [57]** is a hybrid approach that combines the advantages of both IR-based and NMT-based techniques. **Re^{2}Com [52]** is an exemplar-based comment generation approach that leverages the advantages of three types of methods based on neural networks, templates, and IR to improve the performance. **EditSum [30]** is the most recent hybrid approach. It first retrieves the most similar code snippet, and treats the corresponding summary as a prototype. Then, it combines the pattern in the prototype and semantic information of the input code to generate the target summary.

4.1.2 Results. Table 2 shows the comparison results between the performance of DECOM and other baselines, and the best performance is highlighted in bold. Overall, our approach achieves the best performance on all evaluation metrics, followed by Rencos, EditSum, and Re^{2}com. On the Java dataset, DECOM achieves 22.3, 44.5, 19.6, and 2.442 points on BLEU-4, ROUGE-L, METEOR, and CIDEr.

Table 2: The results of comparison with baselines, with the improvement compared with the best baselines in percentage.

Method	JCSD							PCSD						
	BLEU-1/2/3/4				ROUGE-L	METEOR	CIDEr	BLEU-1/2/3/4				ROUGE-L	METEOR	CIDEr
LSI	31.4	22.5	19.3	17.3	34.8	14.4	1.803	36.3	23.6	20.1	17.6	40.0	17.2	1.982
VSM	33.3	24.4	21.1	19.0	36.6	15.4	1.983	38.9	26.1	22.1	19.3	42.7	19.0	2.216
NNGen	33.0	24.4	20.9	18.7	36.3	15.0	1.933	36.5	23.8	20.1	17.4	40.2	17.1	1.967
CODE-NN	23.9	12.8	8.6	6.3	28.9	9.1	0.978	30.8	15.4	10.7	8.1	35.1	13.4	1.229
TL-CodeSum	29.9	21.3	18.1	16.1	33.2	13.7	1.660	31.1	16.5	12.5	10.4	35.3	13.6	1.335
Hybrid-DRL	32.4	22.6	16.3	13.3	26.5	13.5	1.656	41.1	26.2	19.5	15.0	42.2	17.9	2.042
Re^{2}com	33.7	23.6	19.0	16.3	38.1	15.1	1.807	36.6	22.3	17.4	14.5	40.8	17.0	1.813
Rencos	37.5	27.9	23.4	20.6	42.0	17.3	2.209	43.1	29.5	24.2	20.7	47.5	21.1	2.449
EditSum	34.1	24.3	19.5	16.9	38.6	15.2	1.865	37.7	23.1	18.2	15.6	42.0	17.1	1.894
DECOM	**40.4**	**30.2**	**25.2**	**22.3**	**44.5**	**19.6**	**2.442**	**45.6**	**31.4**	**25.5**	**21.9**	**49.3**	**22.5**	**2.603**

Table 3: RQ2 Ablation study on the multi-pass deliberation and evaluation model.

Variants	JCSD							PCSD						
	BLEU-1/2/3/4				ROUGE-L	METEOR	CIDEr	BLEU-1/2/3/4				ROUGE-L	METEOR	CIDEr
DECOM w/o Multi-pass Deliberation	38.9	28.5	23.5	20.8	43.1	18.8	2.274	43.5	29.3	23.8	20.4	47.5	21.1	2.424
DECOM w/o Evaluation Model	39.5	29.3	24.3	21.5	43.7	19.0	2.338	44.6	30.3	24.3	20.6	48.6	21.6	2.478
DECOM	**40.4**	**30.2**	**25.2**	**22.3**	**44.5**	**19.6**	**2.442**	**45.6**	**31.4**	**25.5**	**21.9**	**49.3**	**22.5**	**2.603**

Compared with the best baseline (Rencos), DECOM improves the performance of BLEU-4, ROUGE-L, METEOR, and CIDEr by 8.3%, 6.0%, 13.3%, and 10.5%, respectively. On the Python dataset, DECOM achieves 21.9, 49.3, 22.5, and 2.603 points on BLEU-4, ROUGE-L, METEOR, and CIDEr. Compared with the best baseline (Rencos), DECOM also achieves 5.8%, 3.8%, 6.6%, and 6.3% improvements on BLEU-4, ROUGE-L, METEOR, and CIDEr, respectively .

Answering RQ1: DECOM outperforms the state-of-the-art baselines in terms of all seven metrics on both two datasets. Compared to the best baseline Rencos, DECOM improves the performance of BLEU-4, ROUGE-L, METEOR, and CIDEr by 8.3%, 6.0%, 13.3%, and 10.5% on JCSD dataset, by 5.8%, 3.8%, 6.6%, and 6.3% on PCSD dataset, respectively.

4.2 RQ2: Component Analysis

4.2.1 Variants. To evaluate the contribution of core components, we obtain two variants: (1) **DECOM w/o Multi-pass Deliberation**, which removes the multi-pass deliberation and adopts the one-pass process to generate comments. (2) **DECOM w/o Evaluation Model**, which removes the evaluation model and takes the comment generated by the last (K^{th}) deliberation model as the result. We train the two variants with the same experimental setup as DECOM and evaluate their performance on the test sets of JCSD and PCSD, respectively.

4.2.2 Results. Table 3 presents the performances of DECOM and its two variants. We can see that, removing the two components makes the performance degrade substantially. Specifically, when comparing DECOM and DECOM w/o Multi-pass Deliberation, removing the multi-pass deliberation will lead to a dramatic decrease in the average BLEU-4 (by 6.8%), ROUGE-L (by 3.4%), METEOR (by 5.2%), and CIDEr (by 6.9%) across both datasets. When comparing DECOM and DECOM w/o Evaluation Model, we find that removing the evaluation model will lead to the performance decline in the average BLEU-4 (by 4.8%), ROUGE-L (by 1.6%), METEOR (by 3.5%), and CIDEr (by 4.5%). We can also observe that, removing the multi-pass deliberation will lead to a larger degree of performance decline than removing the evaluation model.

Answering RQ2: Both the multi-pass deliberation and the evaluation model components have positive contributions to the performance of DECOM, where the multi-pass deliberation component contributes more to increasing the performance.

4.3 RQ3: Performance for Different Lengths

4.3.1 Methodology. To answer this question, we analyze the performance of DECOM and best three baselines (i.e. Re^{2}com, Rencos, and EditSum) on different lengths (i.e., number of tokens) of code and comments. We calculate the BLEU-1 score of each sample on the test set of both datasets and average the scores by the length of code and comments, respectively. (Note that, based on our observations, all the seven evaluation metrics show similar trends. For simplicity, we show BLEU-1 only).

4.3.2 Results. Figure 5 presents the performance of DECOM and the three baselines on JCSD and PCSD datasets with code and comments of different lengths, where the red lines denote the performance of DECOM. Overall, we can observe that the performance of DECOM generally outperforms the three baselines with different code and comment lengths on both datasets. Specifically, as the length of the input code increases, DECOM almost keeps a stable improvement over the other three approaches. The performance of DECOM is nearly the best on all the lengths of Java and Python code snippets. In particular, DECOM can achieve much higher performance than others when the length of the Java code snippet is over 200 words. This shows that DECOM can better understand the semantics of the long code snippets by sharing the information between deliberation models and the evaluation model. For the output comments, we can see that when the output comments are becoming complicated with a relatively long length, the performance of all the approaches decrease, which indicates that the longer the

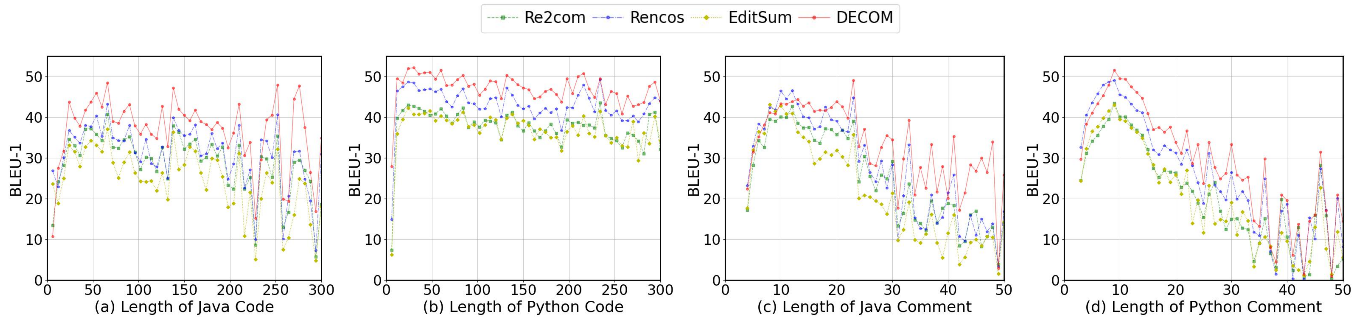

Figure 5: BLEU-1 scores for different code and comment lengths.

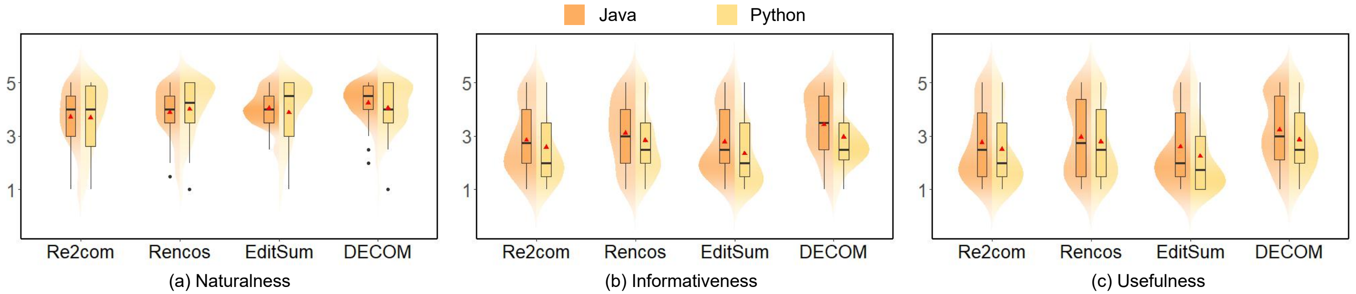

Figure 6: The results of human evaluation.

comment, the harder to generate it completely. However, DECOM still has a substantial improvement over the other baselines (as shown in Figure 5(c)), showing that our approach has the ability to generate long and concise comments.

Answering RQ3: DECOM generally outperforms the best three baselines on different lengths of the input code snippets and the output comments, indicating its robustness. In particular for Java, DECOM can achieve much higher performance than others when the code snippets and comments are long.

5 HUMAN EVALUATION

Although the evaluation metrics (i.e., BLEU, ROUGE-L, METEOR, and CIDEr) can measure the lexical gap between the generated comments and the references, it can hardly reflect the semantic gap. Therefore, we perform a human evaluation to further assess the quality of comments generated by different approaches.

5.1 Procedure

We recruited six participants, including three Ph.D. students, one Master student, and two senior researchers, who are not co-authors of this paper. They all have at least three years of both Java and Python development experience, and four of them have more than six years of development experience. We randomly select 100 code snippets from the test dataset (50 from JCSD and 50 from PCSD). By applying the best three baselines (i.e., Re^{2}com, Rencos, and EditSum) and DECOM, we obtain a total of 400 generated comments. The 400 code-comment pairs are divided into three groups, and each group is used to create a questionnaire. We randomly list the code-comment pairs on the questionnaire and remove their labels to ensure that the participants are not aware of where the comments are generated from. Each questionnaire is evaluated by two participants, and the final result of a generated comment is the average of two participants. Each participant is asked to rate each comment from the three aspects: (1) **Naturalness** reflects the fluency of generated text from the perspective of grammar; (2) **Informativeness** reflects the information richness of generated summaries; and (3) **Usefulness** reflects how can generated summaries help developers. All three scores are integers, ranging from 1 to 5 (5 for excellent, 4 for good, 3 for acceptable, 2 for marginal, and 1 for poor).

Table 4: The statistic results of human evaluation.

	Approach	Avg.	Median	Std.
Naturalness	Re^{2}com	3.7	4.0	1.1
	Rencos	3.9	4.0	1.0
	EditSum	4.0	4.0	0.9
	DECOM	**4.1**	4.5	0.8
Informativeness	Re^{2}com	2.7	2.5	1.3
	Rencos	3.0	3.0	1.3
	EditSum	2.6	2.0	1.3
	DECOM	**3.2**	3.0	1.2
Usefulness	Re^{2}com	2.6	2.0	1.4
	Rencos	2.9	2.5	1.4
	EditSum	2.4	2.0	1.3
	DECOM	**3.1**	3.0	1.3

5.2 Results

Figure 6 exhibits the results of human evaluation by showing the violin plots depicting the naturalness, informativeness, and usefulness of different models, and Table 4 shows the statistic results. Each violin plot contains two parts, i.e., the left and right parts reflect the

evaluation results of models on the JCSD dataset and PCSD dataset. The box plots in the violin plots present the distribution of data and the red triangles mean the average scores of the three aspects. Overall, DECOM is better than all baselines in three aspects. The average score for naturalness, informativeness, and usefulness of our approach are 4.24, 3.43, and 3.25, respectively, on the JCSD dataset. On the PCSD dataset, our approach gets the average score of 4.05, 2.96, and 2.87 in terms of naturalness, informativeness, and usefulness. We can see that, the comments generated in the PCSD dataset receive lower scores in human evaluation, while receiving higher scores in evaluation metrics (see Table 2). This is mainly because the PCSD dataset contains shorter comments (see Table 1), thus mistakenly generating fewer keywords may lead to a lower degree of human satisfaction. While the shorter comments are more probable with these N-gram matching metrics [39].

Specifically, in terms of naturalness, our approach achieves average scores above 4 on both JCSD and PCSD datasets, which shows that DECOM can generate fluent and readable comments. Besides, in terms of informativeness and usefulness, DECOM is the only approach with an average score of more than 3 points on the JCSD dataset. It indicates that the comments generated by DECOM tend to be more informative and useful than other baselines.

6 DISCUSSION

6.1 Qualitative Analysis

For qualitative analysis of our approach, we present two cases generated by the best three baselines together with DECOM. The cases are selected from the test sets of Java and Python datasets respectively, as shown in Figure 7 . Overall, the comments generated by DECOM tend to be more accurate and more readable than the other three baselines. In case 1, the aim of the Java code is to display the contents of an index. The three baselines mistakenly predict the keyword "displays" as "locates", "writes", and "locates", respectively, resulting in the semantics of the generated comments being different from the ground truth. In contrast, the comment generated by DECOM is exactly the same as the ground truth, indicating that our approach can understand the intention of code concisely. In case 2, we can see that, our approach also performs better than other baselines, and the comment generated by DECOM has a high semantic similarity with the ground truth.

We believe that the performance advantage of DECOM mainly comes from two aspects: (1) DECOM can observe the entire previously-generated comment and leverage its global information to polish it. While other baselines can only leverage the previously generated words. (2) DECOM employs an evaluation model that can determine the opportunity when the deliberation process should end, as well as learn the semantic relationship between source code and target comments. Besides, the evaluation model shares its two encoders with the deliberation models, which facilitates the information sharing among these models, and enables DECOM to learn a better representation for code and comments.

6.2 Parameter Analysis

Figure 8 illustrates the impact of the maximum number of deliberations K on the performance of DECOM trained on the PCSD dataset, as well as the time cost (Note that, since the JCSD dataset

Case 1 (Java):

```
public void dumpIndex(boolean showBounds)
                        throws IOException {
  byte ixRecord[]=new byte[SPATIAL_INDEX_RECORD_LENGTH];
  int recNum=0;
  if (shpFileName == null) {
    return;}
  else {
    recNum++;
    int offset=readBEInt(ixRecord,0);
    int length=readBEInt(ixRecord,4);}
  ssx.close();
}
```

Ground Truth: displays the contents of this index .
Re^{2}com: writes in the spatial index file for intersections .
Rencos: locates records in the shape file . the spatial index is searched for intersections.
EditSum: locates records in the shape file .
DECOM: displays the contents of this index .

Case 2 (Python):

```
@pytest.mark.django_db
def test_make_naive_use_tz_false(settings):
  settings.USE_TZ = False
  datetime_object = datetime(2016, 1, 2, 21, 52, 25,
                             tzinfo=pytz.utc)
  assert timezone.is_aware(datetime_object)
  naive_datetime = make_naive(datetime_object)
  assert timezone.is_aware(naive_datetime)
```

Ground Truth: tests datetimes are left intact if use_tz is not in effect .
Re^{2}com: tests datetimes are made naive configured .
Rencos: tests datetimes are made aware of the configured timezone .
EditSum:tests datetimes are made aware intact if timezones is not in admin .
DECOM:tests datetimes are left intact if timezones is not in effect .

Figure 7: Examples of qualitative analysis.

has quite similar results to the PCSD dataset, we only exhibit the results on the PCSD dataset).

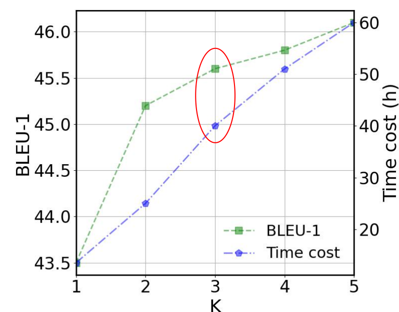

Figure 8: Performance and time cost in varying the maximum number of deliberations K.

We can see that enlarging the maximum number of iterations K generally increases the performance of DECOM. When enlarging K from 1 to 5, the BLEU-1 score increases by 6.0%. We also note that DECOM with $K = 2$ substantially outperforms DECOM with

$K = 1$ (i.e one-pass model), which indicates that the deliberation process can greatly improve the comment quality by polishing the previously generated comment. Moreover, for K larger than 3, the performances slowly increase but the time cost rises exponentially. For example, when enlarging K from 3 to 5, the BLEU-1 score increases by 1% (0.5 points), while the training time increases by 65% (26 hours). Thus, we consider $K = 3$ to be a trade-off choice between effectiveness and efficiency.

6.3 Threats to Validity

There are three main threats to the validity of our approach.

The first threat to validity is that DECOM uses the lexical similarity-based method to retrieve the top similar code-comment pair, which may cause the retrieved comment to be semantically different from the target comment. However, the threat can be mitigated to a large extent, because DECOM generates the target comment in a multi-pass deliberation way, which can iteratively polish the previous comments by leveraging their global information. Thus, even though the dissimilar comment is retrieved, DECOM can correct and refine its content in subsequent iterations, and guarantee the performance is not affected.

The second threat to validity is the datasets we use. We only evaluate DECOM on the Java and Python datasets. However, DECOM uses language-agnostic features that can be easily extracted from any programming language. Therefore, we believe that our approach has good generalizability and can perform well on the datasets of other programming languages, such as C# and Ruby.

The third threat relates to the suitability of evaluation metrics. First, recent researchers have raised concern over the use of BLEU [17], warning the community that the way BLEU is used and interpreted can greatly affect its reliability. To mitigate that threat, we also adopt other metrics, i.e., ROUGE, METEOR, and CIDEr, when evaluating performance. Second, there is also a threat related to our human evaluation. We cannot guarantee that each score assigned to every generated comment is fair. To mitigate this threat, each comment is evaluated by six human evaluators, and we use the average score of the two evaluators as the final score.

The fourth threat relates to the errors in the implementation of baselines. To mitigate this issue, we directly use the publicly available code of CODE-NN, TL-CodeSum, Hybrid-DRL, Rencos, and Re^{2}com to implement baselines. However, the code of EditSum [30] is not available, so we tried our best to understand the paper and re-implement the approach carefully. While we have verified our implementation can achieve similar results as the original EditSum on the same dataset used in its paper.

7 RELATED WORK

7.1 Automatic Comment Generation

The automatic comment generation task now is a rapidly-growing research topic in the community of software engineering and natural language processing.

Early studies typically utilize the template-based approaches and information retrieval (IR) based approaches to generate comments. The basic idea of the template-based approach [36, 37, 45] is to extract the keywords from the code snippets and fill them into the predefined templates. Due to the limitations of manually designing templates, these methods are usually time-consuming and have poor generalization. The IR-based approaches [13, 14, 18, 19, 34, 42, 54, 55] aim to use IR techniques to extract keywords from the source code and compose them into term-based comments for a given code snippet. For example, Edmund et al. [54] generated comment for a given code snippet by retrieving the replicated code samples from the corpus with clone detection techniques. However, the IR-based approaches ignore the semantic relationship between source code and natural language, so the comments they generate are poor readability. Recently, many learning-based methods have been proposed, which train the neural models from a large-scale code-comment corpus to automatically generate comments [11, 24–26, 29, 30, 50–52, 57]. Iyer *et al.* [26] first treated the comment generation task as an end-to-end translation problem and introduced NMT techniques into code comment generation. Hu *et al.* [24] converted the Java methods into AST sequence to learn the structural information, and applied a seq2seq model to generate comments. Zhang *et al.* [57] proposed a seq2seq approach that retrieved two similar code snippets for a given code to improve the quality of the generated comment. Further, Li *et al.* [30] treated the comment of the similar code retrieved from a parallel corpus as a prototype. Based on the semantic differences between input code and similar code, they proposed a seq2seq network to update the prototype and generate comments.

Different from the existing research, we propose a novel framework for automatic comment generation, which performs multiple deliberation processes to iteratively polish the generated comments. Our framework also contains an evaluation model that can not only determine whether to end the deliberation process, but also learn the semantic relationship between source code and target annotations. The experimental results also prove the superiority of our approach.

7.2 Deliberation Networks

The Deliberation mechanism is aimed at polishing the existing results for further improvement. It has been successfully applied to various tasks, such as machine translation [16, 31], image captioning [32], speech recognition [22, 23]. Xia et al. [56] first proposed a deliberation network for sequence generation tasks, which consists of two decoders: a first-pass decoder is used for generating a draft, and a second-pass decoder is used to polish the generated draft to a better sequence. Geng et al. [16] proposed a novel architecture to introduce the deliberation mechanism into the neural machine translation model. It utilizes the policy network to adaptively determine whether to end the translation process. Lian et al. [32] proposed a universal two-pass decoding framework for the image captioning tasks, which contains a drafting model and a deliberation model. The drafting model first generates a draft caption according to an input image, and a deliberation model then refines the draft caption to a better image description. Hu et al. [23] employed the deliberation network for the speech recognition tasks. They combined acoustics and first-pass text hypotheses for second-pass decoding based on the deliberation network and obtained significant improvements.

The findings of previous work motivate the work presented in this paper. Our study is different from the previous work as we

focus on utilizing the deliberation network to enhance the performance of the comment generation task. Besides, we also consider the characteristics of the comment generation task itself: (1) since code reuse is widespread in software development, we use retrieval techniques to retrieve the most similar comment to provide an explicit hint about the comment expression; (2) user-defined identifier names usually contain semantic information, we extract the keywords from the source code to strength the semantic features of the source code. To the best of our knowledge, this is the first work that treats the comment generation process as the process of writing and polishing, and utilizes the multi-pass deliberation automatically generate comments.

8 CONCLUSION

In this paper, we propose a novel multi-pass deliberation framework for automatic comment generation, named DECOM, which is inspired by human cognitive processes. DECOM relies on multiple deliberation models and one evaluation model to iteratively perform the deliberation process. For each process, the deliberation model refines the previously generated comment into a better one. The evaluation model estimates the quality of the new generated comment, and compares its quality score to the previous one to determine whether to end the iterative process. We use a two-step training strategy to train our framework. The evaluation results show that our approach significantly outperforms all other baselines on both Java and Python datasets. A human evaluation study also confirms the comments generated by DECOM tend to be more readable, informative, and useful. In future work, we plan to incorporate the reinforcement learning techniques (e.g. policy network) into the framework to adaptively choose the suitable deliberation processes, thereby enhancing the performance.

ACKNOWLEDGMENTS

We sincerely appreciate anonymous reviewers for their constructive and insightful suggestions for improving this manuscript. This work is supported by the National Key Research and Development Program of China under Grant No. 2018YFB1403400, the National Science Foundation of China under Grant No. 61802374, 62002348, 62072442, 614220920020 and Youth Innovation Promotion Association Chinese Academy of Sciences.

REFERENCES

[1] 2016. Lucene. https://lucene.apache.org/.
[2] 2016. Pytorch Framework. https://pytorch.org/.
[3] 2022. Javalang. https://pypi.org/project/javalang.
[4] 2022. Project Website. https://github.com/ase-decom/ASE22_DECOM.
[5] 2022. Tokenize. https://docs.python.org/2/library/tokenize.html.
[6] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020*. 4998–5007.
[7] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer Normalization. *CoRR* abs/1607.06450 (2016). http://arxiv.org/abs/1607.06450
[8] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization@ACL 2005, Ann Arbor, Michigan, USA, June 29, 2005*, Jade Goldstein, Alon Lavie, Chin-Yew Lin, and Clare R. Voss (Eds.). Association for Computational Linguistics, 65–72. https://aclanthology.org/W05-0909/
[9] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275* (2017).
[10] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the Influence of Identifier Names on Code Quality: An Empirical Study. In *14th European Conference on Software Maintenance and Reengineering, CSMR 2010, 15-18 March 2010, Madrid, Spain*. IEEE Computer Society, 156–165. https://doi.org/10.1109/CSMR.2010.27
[11] Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 826–831. https://doi.org/10.1145/3238147.3240471
[12] Junyan Cheng, Iordanis Fostiropoulos, and Barry W. Boehm. 2021. GN-Transformer: Fusing Sequence and Graph Representation for Improved Code Summarization. *CoRR* abs/2111.08874 (2021). arXiv:2111.08874 https://arxiv.org/abs/2111.08874
[13] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. 1990. Indexing by Latent Semantic Analysis. *J. Am. Soc. Inf. Sci.* 41, 6 (1990), 391–407.
[14] Brian P. Eddy, Jeffrey A. Robinson, Nicholas A. Kraft, and Jeffrey C. Carver. 2013. Evaluating source code summarization techniques: Replication and expansion. In *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*. IEEE Computer Society, 13–22. https://doi.org/10.1109/ICPC.2013.6613829
[15] Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lun Yiu Nie, and Xin Xia. 2021. Code Structure Guided Transformer for Source Code Summarization. *CoRR* abs/2104.09340 (2021). arXiv:2104.09340 https://arxiv.org/abs/2104.09340
[16] Xinwei Geng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2018. Adaptive Multi-pass Decoder for Neural Machine Translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun’ichi Tsujii (Eds.). Association for Computational Linguistics, 523–532. https://doi.org/10.18653/v1/d18-1048
[17] David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. 2020. Code to Comment "Translation": Data, Metrics, Baselining & Evaluation. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. 746–757.
[18] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 223–226. https://doi.org/10.1145/1810295.1810335
[19] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, Giuliano Antoniol, Martin Pinzger, and Elliot J. Chikofsky (Eds.). IEEE Computer Society, 35–44. https://doi.org/10.1109/WCRE.2010.13
[20] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR* abs/1207.0580 (2012). arXiv:1207.0580 http://arxiv.org/abs/1207.0580
[21] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
[22] Ke Hu, Ruoming Pang, Tara N. Sainath, and Trevor Strohman. 2021. Transformer Based Deliberation for Two-Pass Speech Recognition. In *IEEE Spoken Language Technology Workshop, SLT 2021, Shenzhen, China, January 19-22, 2021*. IEEE, 68–74. https://doi.org/10.1109/SLT48900.2021.9383497
[23] Ke Hu, Tara N. Sainath, Ruoming Pang, and Rohit Prabhavalkar. 2020. Deliberation Model Based Two-Pass End-To-End Speech Recognition. In *2020 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2020, Barcelona, Spain, May 4-8, 2020*. IEEE, 7799–7803. https://doi.org/10.1109/ICASSP40776.2020.9053606
[24] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, Foutse Khomh, Chanchal K. Roy, and Janet Siegmund (Eds.). ACM, 200–210. https://doi.org/10.1145/3196321.3196334
[25] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing Source Code with Transferred API Knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. 2269–2275.
[26] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics. https://doi.org/10.18653/v1/p16-1195
[27] He Jiang, Liming Nie, Zeyi Sun, Zhilei Ren, Weiqiang Kong, Tao Zhang, and Xiapu Luo. 2019. ROSF: Leveraging Information Retrieval and Supervised Learning for Recommending Code Snippets. *IEEE Trans. Serv. Comput.* 12, 1 (2019), 34–46. https://doi.org/10.1109/TSC.2016.2592909

[28] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

[29] Alexander LeClair, Aakash Bansal, and Collin McMillan. 2021. Ensemble Models for Neural Source Code Summarization of Subroutines. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021*. IEEE, 286–297. https://doi.org/10.1109/ICSME52107.2021.00032

[30] Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. 2021. EditSum: A Retrieve-and-Edit Framework for Source Code Summarization. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 155–166. https://doi.org/10.1109/ASE51524.2021.9678724

[31] Yangming Li and Kaisheng Yao. 2020. Rewriter-Evaluator Framework for Neural Machine Translation. *CoRR* abs/2012.05414 (2020). arXiv:2012.05414 https://arxiv.org/abs/2012.05414

[32] Zheng Lian, Yanan Zhang, Haichang Li, Rui Wang, and Xiaohui Hu. 2021. Cross Modification Attention Based Deliberation Model for Image Captioning. *CoRR* abs/2109.08411 (2021). arXiv:2109.08411 https://arxiv.org/abs/2109.08411

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

[34] Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: how far are we?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 373–384.

[35] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. *CoRR* abs/2203.07722 (2022). https://doi.org/10.48550/arXiv.2203.07722 arXiv:2203.07722

[36] Paul W. McBurney and Collin McMillan. 2016. Automatic Source Code Summarization of Context for Java Methods. *IEEE Trans. Software Eng.* 42, 2 (2016), 103–119. https://doi.org/10.1109/TSE.2015.2465386

[37] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori L. Pollock, and K. Vijay-Shanker. 2013. Automatic generation of natural language summaries for Java classes. In *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*. IEEE Computer Society, 23–32. https://doi.org/10.1109/ICPC.2013.6613830

[38] 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, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 311–318. https://doi.org/10.3115/1073083.1073135

[39] Ehud Reiter. 2018. A structured Review of the Validity of BLEU. *Computational Linguistics* 44, 3 (2018), 393–401.

[40] Stephen E. Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 3, 4 (2009), 333–389. https://doi.org/10.1561/1500000019

[41] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Justin Gottschlich and Alvin Cheung (Eds.). ACM, 31–41. https://doi.org/10.1145/3211346.3211353

[42] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A Vector Space Model for Automatic Indexing. *Commun. ACM* 18, 11 (1975), 613–620.

[43] Andrea Schankin, Annika Berger, Daniel V. Holt, Johannes C. Hofmeister, Till Riedel, and Michael Beigl. 2018. Descriptive compound identifier names improve source code comprehension. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*. ACM, 31–40. https://doi.org/10.1145/3196321.3196332

[44] Andrew I. Schein, Alexandrin Popescul, Lyle H. Ungar, and David M. Pennock. 2002. Methods and metrics for cold-start recommendations. In *SIGIR 2002: Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 11-15, 2002, Tampere, Finland*, Kalervo Järvelin, Micheline Beaulieu, Ricardo A. Baeza-Yates, and Sung-Hyon Myaeng (Eds.). ACM, 253–260. https://doi.org/10.1145/564376.564421

[45] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto (Eds.). ACM, 43–52. https://doi.org/10.1145/1858996.1859006

[46] Yifan Sun, Changmao Cheng, Yuhan Zhang, Chi Zhang, Liang Zheng, Zhongdao Wang, and Yichen Wei. 2020. Circle Loss: A Unified Perspective of Pair Similarity Optimization. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*. Computer Vision Foundation / IEEE, 6397–6406. https://doi.org/10.1109/CVPR42600.2020.00643

[47] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. 1996. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Program. Lang.* 4, 3 (1996), 143–167. http://compscinet.dcs.kcl.ac.uk/JP/jp040302.abs.html

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 5998–6008.

[49] Ramakrishna Vedantam, C. Lawrence Zitnick, and Devi Parikh. 2015. CIDEr: Consensus-based image description evaluation. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 4566–4575. https://doi.org/10.1109/CVPR.2015.7299087

[50] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 397–407.

[51] Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. 2021. Context-aware Retrieval-based Deep Commit Message Generation. *ACM Trans. Softw. Eng. Methodol.* 30, 4 (2021), 56:1–56:30. https://doi.org/10.1145/3464689

[52] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and Refine: Exemplar-based Neural Comment Generation. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. 349–360.

[53] Sam Wiseman and Alexander M. Rush. 2016. Sequence-to-Sequence Learning as Beam-Search Optimization. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, Jian Su, Xavier Carreras, and Kevin Duh (Eds.). The Association for Computational Linguistics, 1296–1306. https://doi.org/10.18653/v1/d16-1137

[54] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. CloCom: Mining existing source code for automatic comment generation. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*. IEEE Computer Society, 380–389. https://doi.org/10.1109/SANER.2015.7081848

[55] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. AutoComment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. IEEE, 562–567. https://doi.org/10.1109/ASE.2013.6693113

[56] Yingce Xia, Fei Tian, Lijun Wu, Jianxin Lin, Tao Qin, Nenghai Yu, and Tie-Yan Liu. 2017. Deliberation Networks: Sequence Generation Beyond One-Pass Decoding. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 1784–1794. https://proceedings.neurips.cc/paper/2017/hash/c6036a69be21cb660499b75718a3ef24-Abstract.html

[57] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1385–1397. https://doi.org/10.1145/3377811.3380383

[58] Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald C. Gall. 2021. Adversarial Robustness of Deep Code Comment Generation. *CoRR* abs/2108.00213 (2021). arXiv:2108.00213 https://arxiv.org/abs/2108.00213