Automating Just-In-Time Comment Updating

Zhongxin Liu[*†]
Zhejiang University
China
liu_zx@zju.edu.cn

Xin Xia[‡]
Monash University
Australia
xin.xia@monash.edu

Meng Yan
Chongqing University
China
mengy@cqu.edu.cn

Shanping Li
Zhejiang University
China
shan@zju.edu.cn

ABSTRACT

Code comments are valuable for program comprehension and software maintenance, and also require maintenance with code evolution. However, when changing code, developers sometimes neglect updating the related comments, bringing in inconsistent or obsolete comments (aka., bad comments). Such comments are detrimental since they may mislead developers and lead to future bugs. Therefore, it is necessary to fix and avoid bad comments. In this work, we argue that bad comments can be reduced and even avoided by automatically performing comment updates with code changes. We refer to this task as "Just-In-Time (JIT) Comment Updating" and propose an approach named **CUP** (**C**omment **UP**dater) to automate this task. CUP can be used to assist developers in updating comments during code changes and can consequently help avoid the introduction of bad comments. Specifically, CUP leverages a novel neural sequence-to-sequence model to learn comment update patterns from extant code-comment co-changes and can automatically generate a new comment based on its corresponding old comment and code change. Several customized enhancements, such as a special tokenizer and a novel co-attention mechanism, are introduced in CUP by us to handle the characteristics of this task. We build a dataset with over 108K comment-code co-change samples and evaluate CUP on it. The evaluation results show that CUP outperforms an information-retrieval-based and a rule-based baselines by substantial margins, and can reduce developers' edits required for JIT comment updating. In addition, the comments generated by our approach are identical to those updated by developers in 1612 (16.7%) test samples, 7 times more than the best-performing baseline.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; *Maintaining software*; Software evolution.

KEYWORDS

Comment updating, Code-comment co-evolution, Seq2seq model

ACM Reference Format:
Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. 2020. Automating Just-In-Time Comment Updating. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416581

1 INTRODUCTION

Code comments are a vital source of software documentation. Developers record various information, such as the intention, implementation and usage of a code segment, code relations, and code evolutions [34, 36, 42] in comments, which makes code comments valuable for understanding source code and facilitating the communication between developers [53, 55]. A prior study has shown that, besides source code, comments are considered as the most essential software artifacts for program comprehension and maintenance [7].

Despite the value of comments, developers may forget or ignore the updates of comments when changing source code [43, 50], which may bring in inconsistent or obsolete comments (aka., bad comments) [43, 44, 50]. Table 1 presents a bad comment example in Apache Kafka [1]. The method in Table 1 registers the metrics of the producer, consumer, and admin clients, while only the consumer clients are mentioned in its associated comment. Hence, the comment is an inconsistent comment. Before being fixed by developers, this comment had existed for over eight months. During this time, it may mislead developers, waste their time to double-check the implementation, complicate code reviews, and result in the introduction of bugs [16, 35, 43, 44, 46]. Therefore, bad comments have negative effects to the robustness of a system and may increase the cost of its development and maintenance. It is necessary to fix bad comments in time or avoid introducing them.

To figure out how and when bad comments are introduced, we further checked the change history of the method in Table 1. We found that at the beginning, only the consumer clients' metrics were registered in this method, but two following code changes added the producer and admin clients' metrics, respectively, without

[*] Also with Ningbo Research Institute.
[†] Also with PengCheng Laboratory.
[‡] Corresponding author.

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416581

Table 1: A bad comment example

```
public Map<MetricName, ? extends Metric> metrics() {
    ...
    for (final StreamThread thread : threads) {
        result.putAll(thread.producerMetrics());
        result.putAll(thread.consumerMetrics());
        result.putAll(thread.adminClientMetrics());
    }
    ...}
```

Method Comment: Get read-only handle on global metrics registry, including streams client's own metrics plus its embedded consumer clients' metrics.

Updated Comment: Get read-only handle on global metrics registry, including streams client's own metrics plus its embedded producer, consumer and admin clients' metrics.

updating the comment. This finding inspires us that if comments can be automatically updated with each code change, it is possible to reduce and even avoid the introduction of bad comments.

In this work, we refer to the task that performs comment updates with code changes as "Just-In-Time (JIT) Comment Updating" and propose a novel approach named **CUP** (**C**omment **UP**dater) to automate this task. Intuitively, this task can be automated using manually derived patterns and rules. However, comments are free-form texts written in natural languages and are far less formal than source code. Thus, it is challenging and time-consuming to manually summarize and adaptively apply comment update patterns. CUP tackles this task in another way. It leverages a novel neural sequence-to-sequence (seq2seq) model to learn the patterns of comment updates occurring with code changes and automatically generate new comments based on the corresponding old comments and code changes. CUP can be used to assist developers in updating comments during code changes, and can consequently help reduce and avoid the introduction of bad comments.

Neural seq2seq models have been shown to be effective for many software engineering (SE) tasks [14, 28, 48]. However, the seq2seq models used in other tasks cannot be directly adopted to JIT comment updating due to two main characteristics of this task: First, we need to preserve the format of comments while dealing with out-of-vocabulary (OOV) words. OOV words are pervasive in software artifacts and need to be carefully handled in many SE tasks [14, 28, 48]. Furthermore, because the goal of this task is to update comments instead of generate them from scratch, it is also necessary to keep the format of old and new comments consistent. Second, this task takes both code changes and old comments as input. Old comments serve as the basis of updates and code changes can provide important guidance and clues. Thus, this task requires neural seq2seq models to learn the representations of code changes and old comments simultaneously and capture their relationships effectively.

To cope with the first characteristic, we propose a simple but effective way to tokenize code and comments, which can not only reduce OOV words but also keep the format information of comments. The copy mechanism [38] is also adopted to copy OOV words and format information from the input during generation. For the second characteristic, our seq2seq model first leverages two distinct encoders to encode code changes and old comments, respectively. Then, to better capture relationships between code changes and comments, we build a unified vocabulary for both code and comment tokens, adopt a pre-trained fastText model to obtain word embeddings, and integrate a novel co-attention mechanism to our seq2seq model. The unified vocabulary ensures the same tokens appearing in code and comments have identical representations. The fastText embeddings provide accurate syntactic and semantic information of each token. The co-attention mechanism can effectively link and fuse information in code changes and comments.

To evaluate our approach, we extract code changes from 1,496 popular engineered Java projects hosted on GitHub, carefully constructing a dataset with 108K code-comment co-change samples. An information-retrieval-based (IR-based) method, a rule-based method and a special method which directly outputs old comments are used as baselines. We evaluate CUP and the baselines on our dataset in terms of Accuracy, Recall@5 and two metrics proposed by us named Average Edit Distance (AED) and Relative Edit Distance (RED). The evaluation results show that CUP outperforms all baselines in terms of all metrics. Specifically, CUP can replicate comment updates performed by developers in 1612 (16.7%) cases, which are 7 times more than the best-performing baseline. It is also the only approach with RED less than 1, which indicates that it can reduce developers' efforts in JIT comment updating.

In summary, the contributions of this paper include:

(1) We propose a novel approach, namely CUP, to automate JIT comment updating. CUP is based on a neural seq2seq model and introduces several customized improvements to effectively handle the characteristics of this task.
(2) We build a dataset with over 108K code-comment co-change samples for JIT comment updating. To the best of our knowledge, it is the first large dataset for this task.
(3) We extensively evaluate CUP on the dataset using four metrics. CUP is shown to outperform three baselines and can reduce developers' efforts in updating comments.
(4) We open source our replication package [4, 5], including the dataset, the source code of CUP, our trained model and test results, for follow-up works.

The remainder of this paper is organized as follows: Section 2 describes the JIT comment updating task and the usage scenarios of our approach. We elaborate on our approach in Section 3 and illustrate how we build our dataset in Section 4. Section 5 presents the procedures and results of our evaluation. In Section 6, we discuss the situations where our approach may fail, a quality assurance method for our approach, the performance of our approach in terms BLEU-4 and METEOR and the threats to validity. After a brief review of related work in Section 7, we conclude this paper and point out future work in Section 8.

2 PROBLEM AND USAGE SCENARIO

In this section, we formalize the JIT comment updating task and describe the usage scenarios of our approach.

2.1 Problem Formulation

This work targets at automating JIT comment updating, i.e., automatically updating comments with code changes. This task can

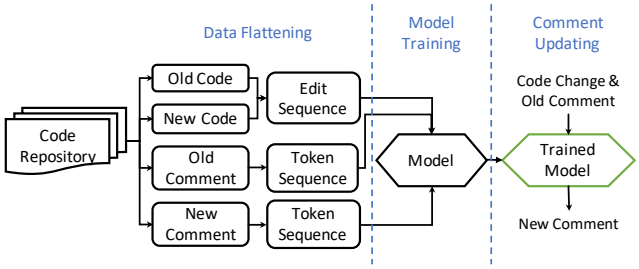

Figure 1: The overall framework of our approach.

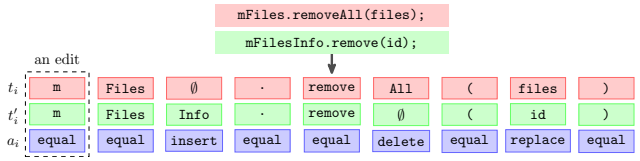

Figure 2: Converting a code change to an edit sequence.

be formalized as follows: given the pre- and post-change versions of a code snippet t, t' and the pre- and post-change versions of its associated comment x, y ($x \neq y$), find a function f so that $f(t, t', x) = y$. Hereon, we refer to t, t', x and y as *old code*, *new code*, *old comment*, and *new comment*, respectively. We tackle this task by devising and training a neural seq2seq model to approximate f. In addition, since the goal is to update comments instead of generate them from scratch, keeping the format of old and new comments consistent is regarded as an essential requirement.

2.2 Usage Scenario

Our approach, namely CUP, takes a code change and its associated old comment as input, aiming to generate the corresponding new comment. Its usage scenarios are as follows:

First of all, CUP can be used to assist developers in performing JIT comment updating. When developers make a code change, CUP can automatically provide update suggestions for the associated comments. If the comments generated by CUP are correct, developers can quickly perform comment updates through one click. Even if CUP's suggestions are only partially correct, they can also reduce developers' edits required to update comments. Therefore, CUP can help improve developers' productivity with respect to JIT comment updating and avoid the introduction of bad comments.

CUP is also able to fix existing bad comments with the help of bad comment detection tools. For example, developers can first leverage the tool proposed by Liu et al. [26] to identify comments requiring updates in each historical code change. Then, CUP can be used to automatically update the detected bad comments instead of manually check and modify them.

3 APPROACH

The overall framework of our approach is illustrated in Figure 1. It consists of three phases, i.e., data flattening, model training, and comment updating. Specifically, we first flatten the code-comment co-change samples extracted from source code repositories as sequences. Then, our neural seq2seq model is trained using the flattened data. Finally, given a code change and its associated old comment, the trained model can automatically generate a new comment to replace the old one. In this section, we elaborate on the data flattening phase and our neural seq2seq model.

3.1 Data Flattening

In this phase, we convert code changes and comments into sequences so that they can be processed by our neural seq2seq model.

3.1.1 Tokenization. For comments, we first tokenize them by spaces and punctuation marks. Spaces are removed while punctuation marks are reserved. Then, compound words, which refer to the tokens constructed by concatenating multiple vocabulary words according to camel or snake conventions, are split into multiple tokens to reduce OOV words. After that, if two adjacent tokens are not split by space, we insert a special token "<con>" between them to mark they are concatenated.

As for code changes, each of them is composed of an old code snippet and a new code snippet. The two snippets are first tokenized using a lexer. Inner comments and white spaces are removed. Each identifier is tokenized based on camel casing and snake casing, and "<con>" is also inserted to join the subtokens. String literals are tokenized like comments.

The key issue in software artifact tokenization is how to deal with compound words. In the literature, the common ways include: not changing compound words [28], splitting them [14] and adding a special symbol "</t>" at the end of each token before splitting [19] (e.g., "inputBuffer" → "inputBuffer</t>" → "input Buffer</t>"). However, the first way cannot reduce OOV words. The second way may lose format information, i.e., a token sequence may fail to be recovered to its original sentence. The third way cannot handle the situation where a subtoken of a compound word is generated as an independent token. For example, "input" cannot be generated as an independent word if it is copied from "inputBuffer", since it does not end with "</t>". Compared to these methods, our tokenizer can be regarded as "asking" the neural model to also learn format information by inserting "<con>" to mark concatenation. Simple is it, it can effectively keep comment format consistent. Also, to preserve format information, we do not lowercase tokens in both code and comments.

3.1.2 Code Change Representation. After tokenization, each code change is converted to two token sequences. We can simply use two encoders to encode them, which however, makes it hard to capture fine-grained modifications between them. To better represent each code change, we first align its two token sequences using a diff tool and then construct an edit sequence based on the alignment, similar to [54], as its representation, as shown in Figure 2. Each element in an edit sequence is a triple $<t_i, t'_i, a_i>$ and is named as an *edit*. t_i is a token in the old code and t'_i is a token in the new code. a_i is the edit action that converts t_i to t'_i, which can be *insert*, *delete*, *equal* or *replace*. If a_i is *insert* (*delete*), t_i (t'_i) will be the empty token $\emptyset$. Such edit sequences can not only preserve the information of the old code and the new code, but also highlight the fine-grained changes between them.

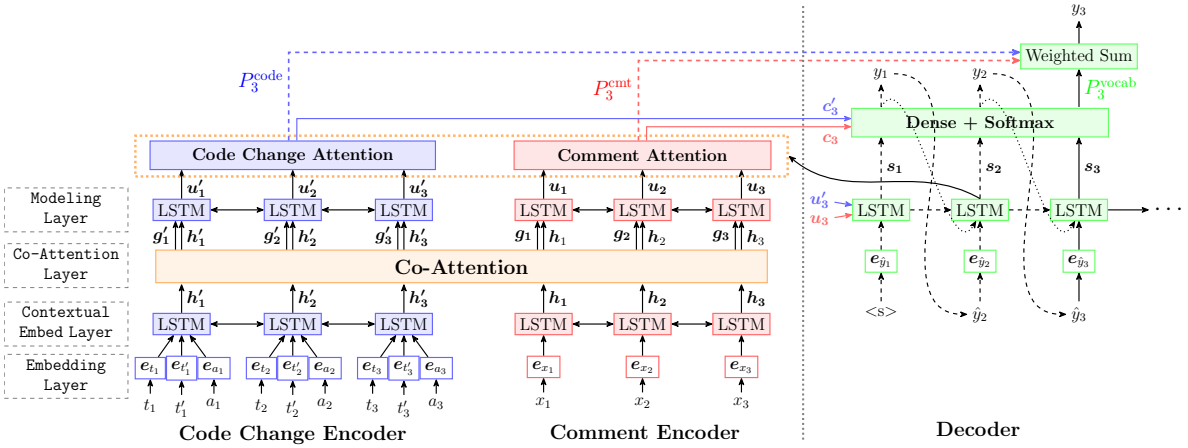

Figure 3: The architecture of our neural seq2seq model.

3.2 Overview of Our Neural Seq2Seq Model

The architecture of our neural seq2seq model is presented in Figure 3. Our model takes as input an edit sequence $\boldsymbol{E} = [<t_1, t'_1, a_1>, \cdots, <|t_n, t'_n, a_n>]$ and an old comment $\boldsymbol{x} = [x_1, \cdots, x_{|\boldsymbol{x}|}]$, aiming to generate the new comment $\boldsymbol{y} = [y_1, \cdots, y_{|\boldsymbol{y}|}]$. n is the length of the edit sequence. In detail, it leverages two distinct encoders, i.e., Code Change Encoder and Comment Encoder, to encode the edit sequence and the old comment, respectively, and generates the new comment through an LSTM (long short-term memory) [13] decoder. An encoder-side co-attention mechanism is leveraged to learn the relationships between the code change and the old comment. Two pointer generators [38] are used in the decoder to enable copying tokens from both the new code and the old comment during generation.

3.3 Encoders

The two encoders, i.e., Code Change Encoder and Comment Encoder, are nearly the same in structure. Each encoder is composed of four ordered layers: an embedding layer, a contextual embed layer, a co-attention layer, and a modeling layer.

3.3.1 The Embedding Layer. This layer is responsible for mapping the three kinds of tokens, i.e., code tokens, comment tokens, and edit actions, into embeddings. There are only four edit actions, so we randomly initialize an embedding matrix for them and update it during training. For code and comment tokens, we first build a unified vocabulary from all training code and comment tokens. Then we use a pre-trained fastText model [11] to obtain the word embedding of each token. Instead of two distinct vocabularies for code and comments, we prefer a unified one because it ensures the same tokens in code and comments have the same embeddings, which can ease the capture of references between code and comments. Pre-trained word embeddings are used for providing accurate syntactic and semantic information. In addition, we choose fastText instead of other pre-trained models because the word embeddings learned by fastText contain subtoken information and it can effectively map OOV words and subtokens into embeddings, which are very suitable for this task.

3.3.2 The Contextual Embed Layer. For each encoder, we place a distinct Bi-LSTM (Bidirectional LSTM) on the top of the embedding layer to model the temporal interactions between edits (comment tokens) and represent each edit (comment token) as a contextual vector. For Code Change Encoder, the three embeddings, i.e., $\boldsymbol{e}_{t_i}, \boldsymbol{e}_{t'_i}, \boldsymbol{e}_{a_i}$, of each edit $\boldsymbol{E_i}$ are first concatenated horizontally, and then input to the Bi-LSTM, as follows:

$$\boldsymbol{h'_i} = \text{Bi-LSTM}(\boldsymbol{h'_{i-1}}, \boldsymbol{h'_{i+1}}, [\boldsymbol{e}_{t_i}; \boldsymbol{e}_{t'_i}; \boldsymbol{e}_{a_i}])$$

where $\boldsymbol{h'_i}$ is the contextual vector of this edit. Comment Encoder computes the contextual vector $\boldsymbol{h_i}$ of each comment token x_i in a similar way with x_i's embedding $\boldsymbol{e}_{x_i}$ as input. For convenience, the contextual vectors of the old comment and the code change can be stacked as matrices $\boldsymbol{H} \in \mathbb{R}^{2d \times |\boldsymbol{x}|}$ and $\boldsymbol{H'} \in \mathbb{R}^{2d \times n}$, respectively.

3.3.3 The Co-Attention Layer. So far, the code change and the old comment are represented independently. However, to capture relationships between them, it is necessary to link and fuse their information. This layer is used to address this need and is shared by the two encoders. It takes as input the contextual vectors, i.e., $\boldsymbol{H}$ and $\boldsymbol{H'}$, and outputs a comment-aware (edit-aware) feature vector for each edit (comment token) along with the original contextual vector of this edit (comment token) to the consequent layer.

In detail, each feature vector is indeed a context vector computed by the dot-production attention mechanism [29]. Formally, the feature vector $\boldsymbol{g_i}$ of the comment token x_i is calculated by:

$$\boldsymbol{g_i} = \boldsymbol{H'}\boldsymbol{\beta_i} \tag{1}$$

$$\boldsymbol{\beta_i} = \text{softmax}(\boldsymbol{H'}^{\top}\boldsymbol{W_\beta}\boldsymbol{h_i}) \tag{2}$$

$\boldsymbol{\beta_i}$ is the attention weights of x_i on all edits and measures how important each edit is with respect to x_i. $\boldsymbol{W_\beta} \in \mathbb{R}^{2d \times 2d}$ is the trainable parameters. The feature vector $\boldsymbol{g'_i}$ of the edit $\boldsymbol{E_i}$ is computed in nearly the same way except that the attentions are derived oppositely, i.e., from edits to comment tokens, as follows:

$$\boldsymbol{g'_i} = \boldsymbol{H}\boldsymbol{\beta'_i}$$

$$\boldsymbol{\beta'_i} = \text{softmax}(\boldsymbol{H}^{\top}\boldsymbol{W_\beta}^{\top}\boldsymbol{h'_i})$$

We can see that $\boldsymbol{g_i}$ signifies and captures the information related to comment token x_i from the whole code change. Meanwhile, $\boldsymbol{g'_i}$ highlights and keeps the information related to edit $\boldsymbol{E_i}$ from the whole old comment. These feature vectors provide a foundation for capturing relationships between code and comments.

3.3.4 The Modeling Layer. This layer produces the final representation of each edit (comment token) based on its contextual vector and comment-aware (edit-aware) feature vector. The two encoders use two distinct Bi-LSTMs to learn such representations. In detail, given a comment token x_i, its final representation $\boldsymbol{u_i}$ is calculated as follows:

$$\boldsymbol{u_i} = \text{Bi-LSTM}(\text{u}_{\boldsymbol{i-1}}, \boldsymbol{u_{i+1}}, [\boldsymbol{g_i}; \boldsymbol{h_i}])$$

The final representation $\boldsymbol{u'_i}$ of an edit $\boldsymbol{E_i}$ is calculated similarly:

$$\boldsymbol{u'_i} = \text{Bi-LSTM}(\text{u}'_{\boldsymbol{i-1}}, \boldsymbol{u'_{i+1}}, [\boldsymbol{g'_i}; \boldsymbol{h'_i}])$$

$\boldsymbol{u_i}$ ($\boldsymbol{u'_i}$) is expected to contain the contextual information of x_i ($\boldsymbol{E_i}$) with respect to both the code change and the old comment. For convenience, we refer to the stacked matrices of all $\boldsymbol{u_i}$ and all $\boldsymbol{u'_i}$ as $\boldsymbol{U} \in \mathbb{R}^{2d \times |\boldsymbol{x}|}$ and $\boldsymbol{U'} \in \mathbb{R}^{2d \times n}$, respectively.

3.4 Decoder

We use an LSTM-based decoder to generate new comments. Bi-LSTM is not suitable for the decoder, since a new comment is generated token by token. The decoder takes as input $\boldsymbol{U}$ and $\boldsymbol{U'}$ obtained from the two encoders and produces the new comment by sequentially generating its tokens.

We concatenate the last hidden states of the two modeling layers as the initial state $\boldsymbol{s_0}$ of the decoder's LSTM. The right side of Figure 3 illustrates how a comment token is generated. In detail, at decoding step j, the input $\hat{y}_j$ is first mapped into an embedding $\boldsymbol{e}_{\hat{y}_j}$ using Comment Encoder's embedding layer. $\hat{y}_j$ is the previous reference token when training or the previous generated token when testing. Then, the decoder computes the hidden state $\boldsymbol{s_j}$ based on $\boldsymbol{e}_{\hat{y}_j}$, the previous hidden state $\boldsymbol{s_{j-1}}$ and the previous output vector $\boldsymbol{o_{j-1}}$ (computed by Equation 3), as follows:

$$\boldsymbol{s_j} = \text{LSTM}(\boldsymbol{s_{j-1}}, [\boldsymbol{e}_{\hat{y}_j}; \boldsymbol{o_{j-1}}])$$

The decoder also adopts the dot-production attention mechanism, which derives a context vector at each time step as the representation of the encoder's input. There are two distinct encoders, so the decoder computes two context vectors, i.e., $\boldsymbol{c_j}$ from the old comment and $\boldsymbol{c'_j}$ from the code change, following Equation 1 and 2.

Then, $\boldsymbol{c_j}$, $\boldsymbol{c'_j}$ and $\boldsymbol{s_j}$ are concatenated to calculate an output vector $\boldsymbol{o_j} \in \mathbb{R}^l$ and a vocabulary distribution P_j^{vocab}:

$$\boldsymbol{o_j} = \tanh(\boldsymbol{V}[\boldsymbol{c_j}; \boldsymbol{c'_j}; \boldsymbol{s_j}]) \tag{3}$$

$$P_j^{\text{vocab}} = \text{softmax}(\boldsymbol{V'o_j})$$

$\boldsymbol{V} \in \mathbb{R}^{(4d+l)\times l}$ and $\boldsymbol{V'} \in \mathbb{R}^{v \times l}$ are learnable parameters and v is the size of the unified vocabulary. P_j^{vocab} can be directly used to generate the target token. For example, we can choose the token with the highest probability as the output of time step j.

However, the decoder cannot generate OOV words if it only chooses tokens from the vocabulary. We observed that an OOV word in a new comment usually can be found in its corresponding old comment and/or new code. Therefore, we also adopt the pointer generator [38] to alleviate the OOV problem following Liu et al. [28]. Specifically, two pointer generators are leveraged to copy tokens from the old comment and the new code, respectively:

$$P_j^{\text{cmt}}(y_j) = \sum_{k:x_k=y_j} \alpha_{jk}$$

$$P_j^{\text{code}}(y_j) = \sum_{k:t'_k=y_j} \alpha'_{jk}$$

$P_j^{\text{cmt}}(y_j)$ and $P_j^{\text{code}}(y_j)$ are the probabilities of copying y_j from the old comment and the new code. α_{jk} and α'_{jk} are the attention weights of x_k and $\boldsymbol{E_k}$ with respect to time step j, and are calculated with the context vectors $\boldsymbol{c_j}$ and $\boldsymbol{c'_j}$

At last, the conditional probability of producing y_j at time step j is computed as:

$$\begin{aligned} p(y_j|\boldsymbol{y}_{<j}, \boldsymbol{x}, \boldsymbol{E}) = \gamma_j P_j^{\text{vocab}}(y_j) + \\ (1-\gamma_j)(\theta_j P_j^{\text{cmt}}(y_j) + (1-\theta_j) P_j^{\text{code}}(y_j)) \end{aligned} \tag{4}$$

γ_j and θ_j measure the probabilities of generating y_j by selecting from the vocabulary and copying from the old comment, respectively. Each of them is modelled by a single-layer feed-forward neural network jointly trained with the decoder.

4 DATA PREPARATION

In this work, we build our dataset from Java programs. However, our approach is language-agnostic and we believe it can be easily adapted for other languages. We concentrate on co-changes between methods and their header comments (method comments), because Java methods can be precisely associated with their comments, while it is non-trivial to accurately link comments and code of other granularity, e.g., a statement. In addition, for comments, our approach captures update patterns at sentence level, i.e., updates one comment sentence at a time. This is because 1) it is relatively easy to recognize patterns at a small but coherent granularity [54] and 2) a method comment with multiple sentences can also be updated iteratively. For convenience, in this section we use *comment* to refer to a comment sentence and *doc* for a whole method comment.

This section describes how we extract method-doc co-change instances, i.e., <*old code*, *new code*, *old doc*, *new doc*>, from code repositories, how we convert qualified instances into method-comment co-change samples, i.e., <*old code*, *new code*, *old comment*, *new comment*>, and how we build our dataset.

4.1 Data Collection

Wen et al. [50] collected a list of 1,500 Java repositories from GitHub for studying code-comment inconsistencies. All the repositories were selected based on a rigorous procedure, have no less than 500 commits, and were manually verified by Wen et al. to be popular engineered projects. We reused this list to collect data. In detail, we first cloned the 1,500 repositories from GitHub. However, we found two repositories, i.e., *pig4cloud/pig* and *wyouflf/xUtils*, had been removed from GitHub and two other repositories, i.e., *liferay/liferay-portal* and *JetBrains/MPS*, were too large to be cloned in reasonable time. Therefore, 1,496 repositories were successfully cloned. Then,

we constructed method-doc co-change instances by extracting modified methods and their corresponding docs from each non-merge commit of every repository. Methods and docs were associated using JavaParser [2]. We obtained 1,063K method-doc co-change instances after filtering out the instances with unchanged docs.

4.2 Modified Method Extraction

It is non-trivial to extract modified methods from a commit, since developers may change method signatures. To do this, we first leveraged GumTree [8] to calculate method mappings between two revisions. Then, based on such mappings, we compared the ASTs of each old method and its new version to identify and extract modified methods. Comments were ignored for AST comparisons.

However, GumTree is not designed for matching methods and we found that for short methods and methods with similar bodies, the method mappings extracted by GumTree are not always accurate. To alleviate this problem, we customized GumTree's matching algorithm to better extract method mappings. In detail, GumTree's matching algorithm takes two trees T_1 and T_2 as input and contains two ordered phases: the top-down phase and the bottom-up phase. The top-down phase matches isomorphic subtrees between the two trees and the bottom-up phase tries to find additional mappings in a bottom-up way. We customized GumTree by adding an additional phase named method-matching phase *between* the two phases.

This method-matching phase is based on our observation that if method m_i in T_1 and method m_j in T_2 have the same signature, they usually should be matched. In addition, if m_i's signature is different from that of m_j but they have the same name and no other methods in both T_1 and T_2 use this name, it is very likely that m_j is modified from m_i. Specifically, this phase first collects unmatched *MethodDeclaration* nodes $\mathcal{M}_1$ and $\mathcal{M}_2$ from T_1 and T_2, respectively. Then, for each method m_i in $\mathcal{M}_1$, if only one method m_j in $\mathcal{M}_2$ has the same signature as it, m_i and m_j are matched and removed from $\mathcal{M}_1$ and $\mathcal{M}_2$. After checking method signatures and updating $\mathcal{M}_1$ and $\mathcal{M}_2$, this phase continues to match the remaining methods in $\mathcal{M}_1$ and $\mathcal{M}_2$ with respect to method names in a similar way.

It took over 290 hours to extract modified methods from the 1,496 repositories using 40 cores of Intel Xeon 2.7GHz CPU.

4.3 Data Preprocessing

We preprocessed the 1,063K method-doc co-change instances as follows:

4.3.1 Filter Out Unqualified Instances. This step aims to reduce unrelated information in docs and filter out unqualified instances. We observed that if a doc is a line comment, it is usually a commented annotation instead of a method description. So, we first removed the instances with line comments as docs. A doc can contain a free-form description section and a tag section. Compared to the description section, the tag section is more formal and structured, and can be well handled using rule-based methods. Therefore, we focused on the description section and deleted the tag section in each doc. Then, "@inheritDoc", code snippets and html tags were removed from each doc. The docs containing "(non-Javadoc)" or non-ascii characters were filtered out. In addition, since our approach focuses on comment updating, which requires the old and new comments to be non-empty and different, we filtered out the instances with empty or identical docs. Finally, the instances containing abstract methods were also deleted to reduce method mismatching. After this step, we obtained 242,649 qualified instances.

4.3.2 Construct Co-Change Samples. A doc may contain multiple sentences. This step is responsible for further processing docs and matching sentences between each old doc and its new doc. Before sentence matching, we first replaced emails, urls, references (e.g., "#123") and versions (e.g., "1.2.3") in docs with "EMAIL", "URL", "REF" and "VERSION", respectively, to reduce noise. Next, we split each doc into sentences using NLTK [3], removed the sentences with only punctuation marks and tokenized the remaining sentences using the tokenizer described in Section 3.1.1. Then, given a pair of docs, we calculated the word-level Levenshtein distance [24], which is the minimum word edits (insertions, deletions and substitutions) required to change a sentence into the other, between each old sentence and each new sentence and constructed a distance matrix. Based on this matrix, the old and new sentences are matched in a best-fit way. After that, we filtered out the matched pairs of which the two sentences are identical if ignoring punctuation. If the distance of a pair is larger than the old sentence's length and 5, this pair should be regarded as a rewrite instead of an update. Hence we also filtered out such pairs. Finally, each remaining matched pair was used to construct a method-comment co-change sample, i.e., <*old code*, *new code*, *old comment*, *new comment*>. We can see that one method-doc co-change instance can be split into multiple method-comment co-change samples, which share the code change but have their own sentence pairs. As a result, we constructed 172,745 method-comment co-change samples, which belong to 147,844 method-doc co-change instances.

4.3.3 Set Max Length and Max Distance. Due to the limited memory of GPU and to reduce the training time, we set the max lengths of code edit sequences, old comments, and new comments to be 500, 50, and 50 based on the corresponding 90th quantiles of our dataset. In addition, a comment change is very likely to be a rewrite instead of an update if the absolute or relative edit distance between the old, and new comments is large. The relative edit distance is defined as the absolute distance divided by the old comment's length. We find that the absolute and relative edit distances of 80% samples are no more than 12 and 0.67, respectively. To reduce comment-rewrite samples, we filtered out the samples of which the absolute or relative distances is larger than 12 or 0.67. At last, we obtained 108,695 method-comment co-change samples, which come from 98,553 method-doc co-change instances.

4.4 Data Splitting

The 108,695 samples are extracted from 48,007 commits. A commit may contain duplicate samples since developers may perform systematic or recurring code changes in one commit [20, 33]. So, before splitting the data, we deduplicated samples within each commit to reduce bias, which resulted in the deletion of 2,981 samples. For each project, we sorted its commits in the ascending order of commit creation time, put the first 80% commits into the training set, shuffled the remaining 20% commits and evenly split them into the validation and test set. In this way, we ensure all comment updates in the training set occurred before those in the validation

and test sets. We also noticed that git operations like "cherry-pick", "rebase" and "squash" can introduce duplicate samples among different commits. Therefore, after splitting, duplicate samples between the test (validation) and training sets were also filtered out by us. As a result, our final training, validation and test sets consist of 85,657, 9,475, 9,673 method-comment co-change samples, respectively.

5 EVALUATION

In this section, we first present the baselines and the evaluation metrics. Then, we describe our experiment settings, research questions (RQs), and the corresponding experimental results.

5.1 Baselines

To evaluate the performance of CUP, we use three baselines belonging to different types: Origin, FracoUpdater and NNUpdater.

5.1.1 Origin. Origin is a special baseline which directly outputs the old comments as results. By comparing CUP with Origin, we can know whether the comments generated by CUP are closer to the new comments than the old comments.

5.1.2 FracoUpdater. Fraco [37] is a tool proposed by Ratol and Robillard to detect fragile comments with respect to rename refactorings and is shown to perform better than Eclipse's refactoring tool. Although the paper proposing Fraco does not claim that Fraco can update fragile comments with rename refactorings, we find the implementation of Fraco provides a quick-fix feature to fix detected fragile comments. When developers conduct a rename refactoring, Fraco will be triggered to identify the references between comment phrases and the renamed identifier. The quick-fix feature can then automatically replace fragile comment phrases with the new identifier name based on heuristic rules. We manually extract the detection algorithm and the quick-fix feature from Fraco's source code and wrap them as an offline comment updating tool named FracoUpdater by us. Given a code change and a corresponding old comment, FracoUpdater first leverages RefactoringMiner [47] to detect rename refactorings from the code change. Then, for each detected rename refactoring, it uses Fraco's detection algorithm to identify fragile comment phrases in the old comment with respect to this rename. Finally, Fraco's quick-fix feature is applied to fix detected fragile phrases. If there is no rename refactoring detected, no fragile comment phrase identified or no fix performed by Fraco's quick-fix feature, FracoUpdater outputs the old comment as the result. We use FracoUpdater as a rule-based baseline.

5.1.3 NNUpdater. NNUpdater, short for Nearest-Neighbor-based comment Updater, is an IR-based baseline proposed by us for this task. Like NNGen [27] for commit message generation, the hypothesis behind NNUpdater is that similar code changes may lead to similar or even the same comment changes. Given a test sample, i.e., a code change and its old comment, NNUpdater first finds its most similar training sample and then reuses the new comment of the nearest neighbor as output. Specifically, to measure the similarity sim_{chg} between two code changes, NNUpdater converts each of them to a unified *diff* file, represents *diff* files as tf-idf vectors and calculates the cosine similarity between such vectors. The similarity sim_{cmt} between two old comments are also calculated in the same way. The similarity *sim* between two samples is then defined as: $sim = \alpha \cdot sim_{\text{chg}} + (1-\alpha) \cdot sim_{\text{com}}, 0 \leq \alpha \leq 1$.

5.2 Evaluation Metrics

We use Accuracy, Recall@5 and two metrics proposed by us for this task, namely Average Edit Distance (AED) and Relative Edit Distance (RED), to evaluate CUP and the baselines.

Accuracy and Recall@5 are used to present to what extend an approach can generate *correct comments*. We use *correct comments* to refer to the generated comments which are identical to the reference comments if we ignore the punctuation marks at the end of the comments. In detail, Accuracy is the percentage of the test samples where *correct comments* are generated on the first tries. Recall@5 is similar to Accuracy, but allows the approach to try 5 times.

AED measures the average edits developers need to perform to perfectly update comments after using a JIT comment updater. RED is similar to AED, but measures the average of relative edit distances. Formally, given a test set with N samples, an approach's AED and RED are:

$$AED = \frac{1}{N}\sum_{k=1}^{N} \text{edit\_distance}(\hat{\boldsymbol{y}}^{(k)}, \boldsymbol{y}^{(k)})$$

$$RED = \frac{1}{N}\sum_{k=1}^{N} \frac{\text{edit\_distance}(\hat{\boldsymbol{y}}^{(k)}, \boldsymbol{y}^{(k)})}{\text{edit\_distance}(\boldsymbol{x}^{(k)}, \boldsymbol{y}^{(k)})}$$

where edit_distance is the word-level Levenshtein distance and $\hat{\boldsymbol{y}}^{(k)}$ refers to the comment generated for the k_{th} sample. We can see that if an approach's RED is less than 1, developers can expect to spend less efforts in updating comment after using this approach.

5.3 Experiment Settings

For our approach, 300-dimensional word embeddings are used for edit actions, code tokens and comment tokens. The fastText model is pre-trained on Common Crawl and Wikipedia [6], and the pre-trained word embeddings are frozen during training. The hidden states of the Bi-LSTMs and the LSTM in our model are 256 and 512 dimensions (i.e., d=256 and l=512), respectively. All the LSTMs have only one layer. The unified vocabulary only keeps the tokens appearing more than once, and its size turns out to be 44,578.

Code Change Encoder, Comment Encoder, and Decoder in our model are jointly trained to minimize the cross entropy. During training, we optimize the parameters of our model using Adam [21] with a batch size of 32. A dropout [41] rate of 0.2 is used for all LSTM layers and the dense layer before computing P_j^{vocab}. The model is validated every 500 batches on the validation set using perplexity (the smaller the better) with a batch size of 32. We set the learning rate of Adam to 0.001 and clip the gradients norm by 5. The learning rate is decayed by a factor of 0.5 if the validation perplexity does not decrease for 5 validations and we call this as a trial. We stop training after 5 trials. The model with best (smallest) validation perplexity is used for evaluation. When testing, beam search of width 5 is used to generate comments.

For NNUpdater, we tune its α on the validation set through grid search with 0.1 as the step size and surprisingly find that $\alpha = 0$, i.e., only using code change similarity, can achieve the best Accuracy. So, we set the α in NNUpdater to 0.

Table 2: Comparisons of our approach with each baseline

Approach	Accuracy	Recall@5	AED	RED
Origin	0.0% (0)	/	3.74	1.000
FracoUpdater	2.0% (196)	/	3.76	1.022
NNUpdater	1.3% (125)	1.4%	15.25	7.068
CUP	**16.7% (1612)**	**26.1%**	**3.54**	**0.958**

*The numbers in brackets are the numbers of generated *correct comments*.

5.4 RQ1: The Effectiveness of Our Approach

To investigate the effectiveness of our approach, i.e., CUP, we evaluate it and the baselines on our dataset in terms of Accuracy, Recall@5, AED, and RED. The evaluation results are shown in table 2. Origin and FracoUpdater only generate one candidate for each sample, so their Recall@5 is marked as "/". We can see that CUP outperforms all the baselines in terms of all evaluation metrics. It can correctly update comments in 1672 (16.7%) cases on the first tries, over 7 times more than the best-performing baseline, and can generate *correct comments* for 26.1% of the test samples within 5 attempts.

Large improvements are achieved by CUP over NNUpdater in terms of all metrics. When compared to Origin and FracoUpdater, CUP performs much better on Accuracy and Recall@5, and also outperforms them in terms of AED and RED by substantial margins. We also conduct Wilcoxon signed-rank tests [51] at the confidence level of 95%. The p-values of CUP compared with the three baselines in terms of Accuracy, Recall@5, AED and RED are all less than 0.001, which means the improvements achieved by CUP are statistically significant. These results indicate CUP can update comments more effectively and accurately than the three baselines. In addition, CUP is the only approach of which the AED is less than Origin's AED and the RED is less than 1. This highlights that CUP can reduce the edits developers need to perform for JIT comment updating.

To further figure out the reasons of CUP's better performance, we manually inspect the test results. Based on our inspection, we find that compared to NNUpdater and FracoUpdater, CUP has two major advantages:

First, CUP can learn and apply diverse comment update patterns automatically, while NNUpdater and FracoUpdater are limited to specific types of comment updates. Specifically, NNUpdater relies on repeating new comments between test and training samples to generate *correct comments*. It may work well on some specific cases, but lacks the generalization ability. FracoUpdater is based on manually summarized rules. It can obtain accurate results on identifier-renaming-related comment updates, but cannot handle other types of updates, e.g., updates related to type change. In contrast, CUP leverages a probabilistic model to learn common patterns of JIT comment updates from extant code-comment co-changes. The patterns learned by CUP are more diverse than those of NNUpdater and FracoUpdater, and can cover more samples. For example, Table 3 presents a test sample. In this sample, the developer used a wildcard, i.e., "*_compiler_t", in the old comment to refer to the annotation and the method name. When the annotation and the method name are changed, the wildcard should also be modified accordingly. It is non-trivial to design and implement rules for this kind of cases. Both NNUpdater and FracoUpdater fail to perform the correct update, while CUP succeeds.

Table 3: Test sample 1

Code Change:

```
- @NativeType("shaderc_spvc_compiler_t")
- public static long shaderc_spvc_compiler_initialize(){
-   long __functionAddress = Functions.
     compiler_initialize;
+ @NativeType("shaderc_spvc_context_t")
+ public static long shaderc_spvc_context_create() {
+   long __functionAddress = Functions.context_create;
    return invokeP(__functionAddress);
  }
```

Old Comment: Any function operating on a {@code *_compiler_t} must offer the basic thread-safety guarantee.

New Comment: Any function operating on a {@code *_context_t} must offer the basic thread-safety guarantee.

NNUpdater: Operation fails.

FracoUpdater: Any function operating on a {@code *_compiler_t} must offer the basic thread-safety guarantee.

CUP-co-attn: Any function operating on a {@code *_compiler_create} must offer the basic thread-safety guarantee.

CUP-uni-vocab: Any function operating on a {@code *_compiler_t} must offer the basic thread-safety guarantee.

CUP-fastText: Any function operating on a {@code *_context_context_t} must offer the basic thread-safety guarantee.

CUP: Any function operating on a {@code *_context_t} must offer the basic thread-safety guarantee.

Second, CUP can update semantic references between code and comments. NNUpdater does not take code-comment relationships into consideration. FracoUpdater is able to detect some semantic matching between renamed identifiers and comment phrases, but its quick-fix rules cannot correctly update such matching. Different from them, CUP explicitly adopts some components, such as the co-attention mechanism and the unified vocabulary, to enable our seq2seq model to effectively capture the relationships between code and comments. Based on our manual inspection, CUP can update not only lexical but also semantic references between code and comments with code changes. Table 4 presents an example. We can see that the developer fixed the "createSessionFolder" as "true", hence the corresponding description in the old comment should be removed. NNUpdater and FracoUpdater fail to handle this case, but CUP accurately identifies such description and removes it when generating the new comment.

> In summary, CUP significantly outperforms the three baselines. The RED of CUP indicates that CUP can help developers reduce their efforts in JIT comment updating.

5.5 RQ2: The Effects of Main Components

The key of this task is to effectively capture the relationships and references between code and comments. To meet this need, we

Table 4: Test sample 2

Code Change:

```
- private String getSessionFileName(String
    sessionIdentifier, boolean createSessionFolder)
+ private String getSessionFileName(String
    sessionIdentifier)
  {
-   File sessionFolder = folders.get(sessionIdentifier,
    createSessionFolder);
+   File sessionFolder = folders.get(sessionIdentifier,
    true);
    return new File(sessionFolder, "data").
        getAbsolutePath();
  }
```

Old Comment: If the session folder (folder that contains the file) does not exist and createSessionFolder is true, the folder will be created.

New Comment: If the session folder (folder that contains the file) does not exist, the folder will be created.

NNUpdater: Marker => Point

FracoUpdater: If the session folder (folder that contains the file) does not exist and createSessionFolder is true, the folder will be created.

CUP-co-attn: If the session folder does not exist and createSessionFolder is true, the folder will be created.

CUP-uni-vocab: If the session folder (folder that contains the file) does not exist and createSessionFolder is true, the folder will be created.

CUP-fastText: If the session folder (folder that contains the file) does not exist and createSessionFolder, the folder will be created.

CUP: If the session folder (folder that contains the file) does not exist, the folder will be created.

adopt a co-attention mechanism, build a unified vocabulary, and use the word embeddings pre-trained by fastText for better representing, linking and fusing the information in code changes and comments. In this research question, we compare CUP with its three variants: 1) **CUP-co-attn**, which removes the co-attention layer from CUP, 2) **CUP-uni-vocab**, which uses two distinct vocabularies, instead of a unified vocabulary, for code and comment tokens, respectively, and 3) **CUP-fastText**, which does not use the embeddings pre-trained by fastText. Such comparisons can help us understand the impacts of the three components to CUP's performance. There are some other improvements adopted by CUP, such as the special tokenizer for preserving comment format and the copy mechanism. We do not investigate their effects because some of them are believed by us to be indispensable for this task and the effectiveness of others has been investigated by previous related works.

The results of our comparisons are presented in Table 5. We can see that CUP performs better than the three variants in terms of all metrics. For Accuracy, the improvements achieved by CUP range from 1.3% to 3.8%, and CUP can generate at least 118 more *correct comments* than the variants. Wilcoxon signed-rank tests [51] are also used to check the significance of the performance improvements. All the p-values are less than 0.01, which means CUP significantly outperforms the three variants. These results indicate that the co-attention mechanism, the unified vocabulary and the fastText embeddings are useful and effective for this task.

Table 5: Comparisons of our approach with three variants

Approach	Accuracy	Recall@5	AED	RED
CUP-co-attn	12.9% (1250)	23.6%	3.72	1.046
CUP-uni-vocab	15.4% (1494)	25.3%	3.59	0.989
CUP-fastText	13.6% (1320)	23.5%	3.73	1.057
CUP	**16.7% (1612)**	**26.1%**	**3.54**	**0.958**

To better understand these performance differences, we manually inspect the test results of the three variants. We find that CUP will capture more incorrect code-comment references without the co-attention mechanism, and it may not know how to update comments or may perform inaccurate updates on the right references if the unified vocabulary and the fastText embeddings are replaced. As an example, for sample 1 in Table 3, CUP-co-attn builds an incorrect reference between "t" and "create", CUP-uni-vocab cannot predict the proper update and regards modifying nothing as the best solution, and CUP-fastText successfully captures the reference between "compiler" and "context" but inaccurately generates two consecutive "context" to update "compiler". In the sample presented in Table 4, CUP-co-attn predicts that the code element "createSessionFolder" is related to the comment phrase "(folder that contains the file)", and incorrectly removes such phrase. CUP-uni-vocab cannot handle this case and chooses to update nothing. CUP-fastText only captures part of the reference and makes an inaccurate update by only deleting " is true". These results demonstrate the three components all play an important role in capturing and updating code-comment references.

> In summary, the co-attention mechanism, the unified vocabulary, and the fastText embeddings adopted by CUP are helpful for capturing and updating code-comment references and can boost the effectiveness of CUP.

6 DISCUSSION

In this section, we discuss the situations where CUP may fail, a quality assurance method which can improve CUP's practicability, the performance of CUP in terms of BLEU-4 and METEOR, and the threats to the validity of this work.

6.1 Where Does Our Approach Fail

We carefully inspect a number of randomly selected samples where CUP fails to generate *correct comments* and summarize several bad situations of CUP from them. A common bad situation is that developers only optimize their language expression and the old and new comments are of the same meaning. Such optimizations can be lexical, e.g., fixing a typo, capitalizing the first word and pluralizing a verb, or semantic, i.e., using a better way to express the same meaning, or both. If the pattern of an expression update is rare in the dataset, CUP may not be able to capture and apply it without any clue.

In the second situation, the code changes and old comments do not provide enough information for CUP to infer the corresponding comment updates. For example, in a test sample, the developer

Table 6: The performance of our approach with the QA filter

Approach	Accuracy	Recall@5	AED	RED
CUP	16.7% (1612)	26.1%	3.54	0.958
CUP+QA	**31.8**% (1612)	**38.3%**	**2.98**	**0.920**

added the phrase "(chunked)" in the new comment. However, "chunked" does not appear in the code change or the old comment, and cannot be derived from any code element. Therefore, CUP is not able to perform the update.

Another situation is that the code changes and/or comment updates are too complicated for CUP to perfectly handle. For example, the developer may modify several similar code elements in a code change and update their semantic references in the comment simultaneously. CUP may correctly update some of the comment phrases but not always all. We think there are two main reasons preventing CUP from perfectly handling complicated code-comment co-changes. First, CUP does not leverage program analysis tools like RefactoringMiner to explicitly extract information from code changes. Although CUP can handle many situations without such tools, it may get confused and fail to focus on the important parts when a code change contains many modifications unrelated to the comment update. Second, for each sample, CUP only scans the old comment and generates the new comment once. Therefore, it may be challenging for CUP to correct many comment phrases simultaneously. It would be interesting to address these limitations, but it is beyond the scope of this work.

In addition, it is worth mentioning that some samples may fit multiple situations instead of only one. For example, developers may update multiple code-comment references and optimize the language expression in one comment change.

6.2 Quality Assurance for Our Approach

From our manual inspection, we also find that for the samples where CUP is not capable of performing perfect JIT comment updates, CUP may choose to modify nothing and directly generate the old comments. Based on this finding, we propose a quality assurance filter (QA filter) to improve the practicability of CUP. Specifically, for each sample, the QA filter simply compares the comment generated by CUP with the old comment. If they are identical, the QA filter marks this sample as imperfect (i.e., cannot be perfectly handled by CUP) and removes it.

After using this QA filter, we re-evaluate CUP on our dataset. Table 6 presents the evaluation results. We can see that the QA filter improves CUP in terms of all metrics. In detail, it nearly doubles the Accuracy as the consequence of filtering out 4602 out of 9673 test samples. The AED and RED also decrease by substantial margins. These results indicate that the QA filter can make our approach more useful and accurate in practice and can improve developers' confidence on our approach.

6.3 Other Evaluation Metrics

Prior studies often use BLEU-4 and METEOR, which are flexible in word order, to evaluate comment generation methods. We also compare the performance of CUP and the baselines on our dataset in terms of BLEU-4 and METEOR. The evaluation results are shown in Table 7, where BLEU-4 and METEOR scores are presented as percentage values between 0 and 100. We can see that CUP can obtain better BLEU-4 and METEOR scores than the three baselines. Since BLEU-4 and METEOR are calculated at corpus level, statistical significance is tested using paired bootstrap resampling following [22] with 1000 resamples. All the p-values are less than 0.001, which indicates that our approach significantly outperforms the three baselines in terms of the two metrics.

Table 7: The BLEU-4 and METEOR scores of our approach and the baselines

Approach	#Update	BLEU-4	METEOR
Origin	0/9673	70.2	50.2
FracoUpdater	631/9673	70.5	50.3
NNUpdater	9142/9673	14.3	17.7
CUP	5071/9673	**72.0**	**51.2**

*#**Update** refers to the number of the test samples where the generated comments are different from the old comments.

In detail, CUP improves NNUpdater by large margins. Origin can achieve high BLEU-4 and METEOR scores since it directly outputs the old comments, which are naturally similar to the corresponding new comments updated by developers. The performance of FracoUpdater is close to that of Origin, because in most (9042 out of 9673) cases, FracoUpdater does not perform any update and directly outputs the old comments too. In contrast, CUP performs updates on 5071 (52.4%) samples and significantly outperforms Origin and FracoUpdater in terms of BLEU-4 and METEOR. These results further confirm the better performance of CUP.

6.4 Threats to Validity

One threat to the validity of this work is that our dataset is built only from Java projects and only contains the updates of method comments, which may not be representative of all programming languages and comment types. However, Java is one of the most popular programming languages. Method comments are an important type of comments and are often referred to by developers for program comprehension. Besides, our proposed model is independent of programming languages and comment types. It can be applied to projects of other languages and be trained to generate other types of comments.

Another threat is related to the method mappings we build from commits. Before extracting modified methods from a commit, we use GumTree to match the methods in two revisions. However, some method mappings identified by GumTree are suboptimal. We mitigate this threat by 1) adding a new phase in GumTree to improve its accuracy in method matching, 2) filtering the samples with abstract methods, which are often mismatched. In addition, we manually checked 200 samples in our test set and only found one suboptimal method mapping. Therefore, we believe the threat is limited.

7 RELATED WORK

This section discusses related work concerning code-comment co-evolution, comment generation, inconsistent comment detection.

7.1 Code-Comment Co-Evolution

Prior works have investigated the co-evolution between source code and code comments from different perspectives [9, 10, 16, 18, 25, 50]. For example, Fluri et al. [9, 10] studied how source code and comments co-evolved and found that 90% of the comment changes triggered by code changes were done in the same revision as the associated code changes. They also highlighted that API comments are often adapted retroactively. In addition, Ibrahim et al. [16] investigated the relationship between comment update practice and software bugs in three open-source systems and found abnormal comment update behavior is a good indicator for predicting future bugs. Linares-Vasquez et al. [25] studied how developers documented database usages in method comments and pointed out that the comments of database-related methods are less frequently updated than source code. Recently, Wen et al. [50] conducted a large-scale empirical study, which analyzed the chances that different code change types trigger comment updates and defined a taxonomy of the code-comment inconsistencies fixed by developers.

Different from these studies, our work aims to automatically update comments with code changes. The empirical findings presented by previous studies motivate our work and shed light into the JIT comment updating task.

7.2 Comment Generation

Automatic comment generation techniques may also help developers update comments by directly generating new comments from changed methods. Many previous works proposed to generate code comments using rule-based and IR-based methods [12, 31, 32, 40, 52]. For example, Sridhara et al. [40] proposed an approach to generate comments for Java methods using summary information in source code and manually defined templates. To generate a comment for a code snippet, ColCom [52] first finds similar code snippets from open source projects and then reuses and tailors their comments as output. Recently, more and more researchers leveraged probabilistic models to perform comment generation [14, 15, 17, 23, 49, 57]. For example, Iyer et al. [17] proposed a neural attention model named CODE-NN to generate summaries for C# and SQL snippets. DeepCom, an approach proposed by Hu et al. [14], uses a structure-based traversal (SBT) method to flatten ASTs and combines such flattened sequences with an encoder-decoder model to generate comments for Java methods. In their follow-up work, Hu et al. [15] devised Hybrid-DeepCom to enhance DeepCom by combining source code and the SBT sequences together to generate comments. In a parallel work, LeClair et al. [23] proposed a similar model, which also represents code texts and the SBT sequences using two distinct encoders, for comment generation.

Although our approach can be regarded as generating comments through a seq2seq model, it focuses on updating pre-existing comments instead of generating comments from scratch. Moreover, when updating a comment, our approach considers both the old comment and the corresponding code change instead of only the new code. Therefore, we believe the JIT comment updating problem and the comment generation problem are different.

7.3 Inconsistent Comment Detection

Researchers have investigated the detection of inconsistent comments. Most prior works focused on the comments related to specific code properties or of specific types [39, 43–46, 56]. For example, Tan et al. [43–45] proposed a series of approaches to detect code-comment inconsistencies related to specific programming concepts, such as lock mechanisms [43, 44], function calls [43] and interrupts [45]. Their approaches extract concept-related rules from comments based on NLP techniques and check source code against the extracted rules using static program analysis. Sridhara et al. [39] proposed a technique to identify obsolete TODO comments based on information retrieval, linguistics and semantics. Several studies targeted at general comments and took code changes into consideration [26, 30, 37]. For instance, Ratol and Robillard [37] proposed a rule-based approach named Fraco to detect fragile comments with respect to identifier renaming. Liu et al. [26] leveraged machine learning techniques and 64 manually-crafted features derived from code, comments and code-comment relationships to check whether to update a comment when its associated code is changed.

All these techniques focus on detecting inconsistent or obsolete comments, while our approach targets at automatically updating comments with code changes to avoid the introduction of inconsistent and obsolete comments. We believe our approach is a complement instead of a competitor to these techniques.

8 CONCLUSION AND FUTURE WORK

This work aims to reduce and avoid the introduction of bad comments by automatically updating comments with code changes, i.e., automating Just-In-Time (JIT) comment updating. To tackle this task, we propose an approach named **CUP** (**C**omment **UP**dater), which leverages a novel seq2seq model to learn common patterns of JIT comment updates from extant code-comment co-changes and can automatically generate new comments based on the corresponding old comments and code changes. Several improvements, such as a special tokenizer and a co-attention mechanism, are introduced in CUP to handle the characteristics of this task. Comprehensive experiments on a dataset with over 108K code-comment co-change samples show that CUP outperforms three baselines by significant margins and can reduce the edits that developers perform for JIT comment updating.

In the future, we plan to investigate the effectiveness of CUP in cross-project settings. We also plan to adapt CUP to other code granularity, such as statements, and other comment types, such as inner comments of methods. In addition, it would be an interesting future direction to propose more advanced techniques to address CUP's limitations.

ACKNOWLEDGMENTS

This research was partially supported by the National Key R&D Program of China (No. 2018YFB1003904), NSFC Program (No. 61972339), the Australian Research Council's Discovery Early Career Researcher Award (DECRA) (DE200100021), and Alibaba-Zhejiang University Joint Institute of Frontier Technologies.

REFERENCES

[1] 2020. A commit in Apache Kafka. https://github.com/apache/kafka/commit/9dc76f8872b862ca008562cdcf8cf50524e2eaa3.
[2] 2020. JavaParser. https://javaparser.org/.
[3] 2020. Natural language toolkit NLTK 3.5 documentation. http://www.nltk.org/.
[4] 2020. Our replication package. https://tinyurl.com/jitcomment.
[5] 2020. Our source code on GitHub. https://github.com/tbabm/CUP.
[6] 2020. Word vectors for 157 languages. https://fasttext.cc/docs/en/crawl-vectors.html.
[7] Sergio Cozzetti B de Souza, Nicolas Anquetil, and Káthia M de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information*. 68–75.
[8] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th International Conference on Automated Software Engineering*. 313–324.
[9] Beat Fluri, Michael Wursch, and Harald C Gall. 2007. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. In *Proceedings of the 14th Working Conference on Reverse Engineering*. 70–79.
[10] Beat Fluri, Michael Würsch, Emanuel Giger, and Harald C Gall. 2009. Analyzing the co-evolution of comments and source code. *Software Quality Journal* 17, 4 (2009), 367–394.
[11] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. 2018. Learning Word Vectors for 157 Languages. In *Proceedings of the International Conference on Language Resources and Evaluation*.
[12] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *Proceedings of the 17th Working Conference on Reverse Engineering*. 35–44.
[13] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780.
[14] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th International Conference on Program Comprehension*. 200–210.
[15] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2019. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* (2019), 1–39.
[16] Walid M Ibrahim, Nicolas Bettenburg, Bram Adams, and Ahmed E Hassan. 2012. On the relationship between comment update practices and software bugs. *Journal of Systems and Software* 85, 10 (2012), 2293–2304.
[17] 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*. 2073–2083.
[18] Zhen Ming Jiang and Ahmed E Hassan. 2006. Examining the evolution of code comments in PostgreSQL. In *Proceedings of the International Workshop on Mining Software Repositories*. 179–180.
[19] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code!= Big Vocabulary: Open-Vocabulary Models for Source Code. *CoRR* abs/2003.07914 (2020). https://arxiv.org/abs/2003.07914
[20] Miryung Kim and David Notkin. 2009. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*. 309–319.
[21] Diederik P Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference on Learning Representations*.
[22] Philipp Koehn. 2004. Statistical significance tests for machine translation evaluation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. 388–395.
[23] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering*. 795–806.
[24] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, Vol. 10. 707–710.
[25] Mario Linares-Vásquez, Boyang Li, Christopher Vendome, and Denys Poshyvanyk. 2015. How do developers document database usages in source code?. In *Proceedings of the 30th International Conference on Automated Software Engineering*. 36–41.
[26] Zhiyong Liu, Huanchao Chen, Xiangping Chen, Xiaonan Luo, and Fan Zhou. 2018. Automatic detection of outdated comments during code changes. In *Proceedings of the 42nd Annual Computer Software and Applications Conference*, Vol. 1. 154–163.
[27] 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 International Conference on Automated Software Engineering*. 373–384.
[28] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. 2019. Automatic Generation of Pull Request Descriptions. In *Proceedings of the 34th International Conference on Automated Software Engineering*. 176–188.
[29] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. 1412–1421.
[30] Haroon Malik, Istehad Chowdhury, Hsiao-Ming Tsou, Zhen Ming Jiang, and Ahmed E Hassan. 2008. Understanding the rationale for updating a function's comment. In *Proceedings of the International Conference on Software Maintenance*. 167–176.
[31] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *Proceedings of the 21st International Conference on Program Comprehension*. 23–32.
[32] Najam Nazar, Yan Hu, and He Jiang. 2016. Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology* 31, 5 (2016), 883–909.
[33] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar Al-Kofahi, and Tien N Nguyen. 2010. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd International Conference on Software Engineering*. 315–324.
[34] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. 2009. Listening to programmers Taxonomies and characteristics of comments in operating system code. In *Proceedings of the 31st International Conference on Software Engineering*. 331–341.
[35] David Lorge Parnas. 2011. Precise documentation: The key to better software. In *The Future of Software Engineering*. Springer, 125–148.
[36] Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. 2019. Classifying code comments in Java software systems. *Empirical Software Engineering* (2019), 1–39.
[37] Inderjot Kaur Ratol and Martin P Robillard. 2017. Detecting fragile comments. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. 112–122.
[38] Abigail See, Peter J Liu, and Christopher D Manning. 2017. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. 1073–1083.
[39] Giriprasad Sridhara. 2016. Automatically detecting the up-to-date status of ToDo comments in Java programs. In *Proceedings of the 9th India Software Engineering Conference*. 16–25.
[40] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the International Conference on Automated Software Engineering*. 43–52.
[41] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research* 15, 1, 1929–1958.
[42] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2013. Quality analysis of source code comments. In *Proceedings of the 21st International Conference on Program Comprehension*. 83–92.
[43] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or bad comments?*. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*. 145–158.
[44] Lin Tan, Ding Yuan, and Yuanyuan Zhou. 2007. Hotcomments: how to make program comments more useful?. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*. 1–6.
[45] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering*. 11–20.
[46] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. 2012. @ tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*. 260–269.
[47] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering*. 483–494.
[48] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering*. 25–36.
[49] 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 International Conference on Automated Software Engineering*. 397–407.
[50] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *Proceedings of the 27th International Conference on Program Comprehension*. 53–64.
[51] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in Statistics*. Springer, 196–202.
[52] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. Clocom: Mining existing source code for automatic comment generation. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*. 380–389.
[53] Scott N Woodfield, Hubert E Dunsmore, and Vincent Yun Shen. 1981. The effect of modularization and comments on program comprehension. In *Proceedings of*

the 5th International Conference on Software Engineering. 215–223.

[54] Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L Gaunt. 2018. Learning to represent edits. In *Proceedings of the 7th International Conference on Learning Representations*.

[55] Annie TT Ying, James L Wright, and Steven Abrams. 2005. Source code that talks: an exploration of Eclipse task comments and their implication to repository mining. In *Proceedings of the International Workshop on Mining Software Repositories*. 1–5.

[56] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs documentation and code to detect directive defects. In *Proceedings of the 39th International Conference on Software Engineering*. 27–37.

[57] Yu Zhou, Xin Yan, Wenhua Yang, Taolue Chen, and Zhiqiu Huang. 2019. Augmenting Java method comments generation with context information based on neural networks. *Journal of Systems and Software* 156 (2019), 328–340.