Automated description generation for software patches

Thanh Trong Vu, Tuan-Dung Bui, Thanh-Dat Do, Thu-Trang Nguyen, Hieu Dinh Vo, Son Nguyen *

Faculty of Information Technology, VNU University of Engineering and Technology, Hanoi, Viet Nam

ARTICLE INFO

ABSTRACT

Software patches are pivotal in refining and evolving codebases, addressing bugs, vulnerabilities, and optimizations. Patch descriptions provide detailed accounts of changes, aiding comprehension and collaboration among developers. However, manual description creation poses challenges in terms of time consumption and variations in quality and detail. In this paper, we propose PATCHEXPLAINER, an approach that addresses these challenges by framing patch description generation as a machine translation task. In PATCHEXPLAINER, we leverage explicit representations of critical elements, historical context, and syntactic conventions. Moreover, the translation model in PATCHEXPLAINER is designed with an awareness of description similarity. Particularly, the model is *explicitly* trained to recognize and incorporate similarities present in patch descriptions clustered into groups, improving its ability to generate accurate and consistent descriptions across similar patches. The dual objectives maximize similarity and accurately predict affiliating groups. Our experimental results on a large dataset of real-world software patches show that PATCHEXPLAINER consistently outperforms existing methods, with improvements up to 189% in *BLEU*, 5.7X in *Exact Match* rate, and 154% in *Semantic Similarity*, affirming its effectiveness in generating software patch descriptions.

1. Introduction

Software patches [1–3] are crucial components that facilitate the ongoing refinement and evolution of codebases. These patches are fundamental in addressing various issues within a software system, ranging from addressing bugs and vulnerabilities to optimizations. Patch descriptions play a vital role in articulating the nature and purpose of these modifications, providing a detailed account of the changes introduced, the specific issues addressed, and the methodologies employed in the patches [4]. This descriptive layer not only serves as a documentation tool but also fosters effective communication and collaboration among developers [4,5]. By explaining the '*what*,' '*where*', and '*how*' of code changes, patch descriptions contribute to a comprehensive understanding of the evolving software, enabling developers to navigate through the details of modifications, collaborate seamlessly, and ensure the stability and functionality of the software system throughout its lifecycle.

Manual creation of patch descriptions poses inherent challenges in the software development cycle. The details involved in translating technical code changes into precise, coherent, and informative descriptions present a time-consuming task for developers. This process not only consumes time but also introduces the potential for inconsistencies

and variations in the quality of descriptions across different developers. Moreover, the diverse backgrounds and perspectives of developers may lead to discrepancies in the level of detail and clarity provided in descriptions. Thus, automating patch description generation is an essential yet highly challenging task. Indeed, automated description generation approaches face the following major challenges:

Contextual Understanding of Code Changes: A significant challenge in patch description generation is the understanding of contextual information within code changes. The approaches must accurately capture the specific details of bug fixes introduced in the code. The complicated relationships between code elements, as well as the bug addressed and the operations employed to address it could pose challenges for the approaches to comprehend and express accurately in natural language.

Syntactic and Stylistic Consistency: Ensuring syntactic and stylistic consistency in generated patch descriptions is another major challenge. The approaches must capture the language preferences of individual developers and specific projects. The diversity in language preferences and the limitations in understanding the broader software patching context could affect the description generation approaches' ability to replicate specific language choices accurately.

* Corresponding author.
E-mail addresses: thanhvu@vnu.edu.vn (T.T. Vu), 21020006@vnu.edu.vn (T.-D. Bui), 20020045@vnu.edu.vn (T.-D. Do), trang.nguyen@vnu.edu.vn (T.-T. Nguyen), hieuvd@vnu.edu.vn (H.D. Vo), sonnguyen@vnu.edu.vn (S. Nguyen).

Conceptually, describing software patches can be viewed as a machine translation (MT) task, where the patch code is the source language and the corresponding description is the target language. Recently, MT techniques, particularly neural machine translation (NMT) techniques, have been increasingly applied across various domains in software engineering [6–14]. NMT has found utility in tasks such as code summarization [6,7], code comment/log generation [8,9], and commit message generation [10–13]. Standard NMT models comprise two components [15,16]: an encoder and a decoder. The encoder's role is to initially comprehend the meaning of words in the source language and convert them into an intermediate numeric representation. After that, the decoder generates target words sequentially, drawing on the intermediate representation and preceding words from the generated sequence. In an ideal scenario, NMT models are anticipated to accurately grasp the meaning of an input sequence and subsequently articulate it in the output sequence. The objective of generating similar descriptions for similar software patches is *implicitly* expected to be achieved through this process of encoding and decoding within the NMT model.

In this paper, we introduce PATCHEXPLAINER, a novel automated approach that tackles the challenges in patch description generation. To enhance contextual understanding, our approach is to incorporate explicit representations of critical elements within patch descriptions, conveying information about bugs/vulnerabilities, operational fixing details, and patching scope. Additionally, we consider historical software patching context to employ syntactic and stylistic conventions and enforce consistency.

Furthermore, to foster accuracy in generating patch descriptions by MT models, our approach is to design a patch description generation model as a translation model with the awareness of the similarity between descriptions. Particularly, the model is *explicitly* trained to recognize and incorporate similarities present in various patch descriptions, improving its ability to generate accurate and consistent descriptions across similar patches. The intuition is that instilling the model with the ability to additionally discern the similarity between descriptions can contribute to more precise description generation. In particular, we handle the similarity between the descriptions by clustering the patches with similar descriptions into a group. To implement the idea of a translation model with the awareness of descriptions' similarity, our approach leverages dual objectives: one is the machine translation task's original objective, *correctly generating expected descriptions*, and the other is *accurately predicting the affiliating group containing similar patches reflected in the descriptions*.

We conducted experiments to evaluate the effectiveness of PATCHEXPLAINER for automated patch description generation in comparison to the state-of-the-art approaches [10–13]. Our results show that PATCHEXPLAINER is consistently better than existing methods across different metrics such as *BLEU*, *Exact Match* rate, and *Semantic Similarity*, demonstrating significant improvements in linguistic precision, semantic alignment, and overall coherence in patch descriptions. Particularly, PATCHEXPLAINER significantly outperforms the existing approaches, with relative improvements ranging from **43%** to **189%** in *BLEU*. The *Exact Match* rate of our approach surpasses that of all compared methodologies by **5.7X**, emphasizing its proficiency in generating descriptions that precisely match reference texts. Furthermore, PATCHEXPLAINER achieves considerably better *Semantic Similarity* compared to the state-of-the-art approaches, with improvements ranging from **16%** to an impressive **154%**, affirming its ability to capture the underlying meaning or intent of patches. The consistent and substantial improvements observed across diverse metrics emphasize the robustness and effectiveness of our approach in automating the patch description generation process.

The contributions of this paper are listed as follows:

1. Contextual Representation: A novel method that uses both semantic and conventional contexts for automated patch description generation.


```
1.   static const struct URLProtocol *url_find_protocol(const char *filename)
2.   {
3.       //...
4.       protocols = ffurl_get_protocols(NULL, NULL);
5.   +   if (!protocols)
6.   +       return NULL;
7.       for (i = 0; protocols[i]; i++) {
8.           const URLProtocol *up = protocols[i];
9.           //...
10.  }
11.  const URLProtocol **ffurl_get_protocols(const char *whitelist,
12.                                           const char *blacklist)
13.  {
14.      const URLProtocol **ret;
15.      int i, ret_idx = 0;
16.
17.      ret = av_calloc(FF_ARRAY_ELEMS(url_protocols), sizeof(*ret));
18.      if (!ret)
19.          return NULL;
20.          //...
21.  }
```

File path:
  libavformat/avio.c
Bug type:
  NULL pointer dereference
Patch Description:
  avformat/avio: Fix null pointer dereference in case of memleak

Fig. 1. A patch in project FFmpeg and its description.

2. Novel technique: PATCHEXPLAINER, a novel automated patch description generation approach, is developed based on machine translation with dual objectives, capturing both semantic and conventional information to generate patch description.

3. Extensive experimental results: An extensive experimental evaluation showing the performance of PATCHEXPLAINER over the state-of-the-art methods for patch description generation.

2. Motivating example

2.1. Examples

This section illustrates the task of automated description generation for software patches via an example.

Fig. 1 shows a patch in project FFmpeg[1] and its corresponding description. In this example, function `url_find_protocol` finds and returns a suitable URL protocol for a given `filename`. However, this function could encounter a NULL pointer dereference problem (line 7) if the list of protocols, `protocols`, returned by function `ffurl_get_protocols` is null (line 4). To avoid this problem, an `if` statement was added (lines 5–6) to prevent this function from accessing a null `protocols` list. The author of the commit, *Michael Niedermayer*, described the patch as *"avformat/avio: Fix null pointer dereference in case of memleak"*.

The description provides descriptive information about the patch: ***what***, ***where***, and ***how*** the patch is fixing. In Fig. 1, the patch was introduced to fix a *NULL pointer dereference* issue in `avio.c` of module `avioformat` by preventing the use of null `protocols` to handle *memory leak* cases. To understand better the *what*, *where*, and *how* aspects in the real-world patch descriptions, we performed a rule-based analysis, which has been used in the existing studies on change to text [17,18], on 24.2 K patches and their corresponding description collected from the datasets published by the existing studies [19]. Additionally, we manually analyzed a set of randomly sampled patches to carefully investigate these aspects and minimize the threats to the validity of the rule-based analysis. Specifically, for the description *d* of a patch *p*, we applied the rule-based analysis to determine if *d* provides a certain aspect:

1. To determine whether the description contains the information about *what* the patch is fixing bugs, we investigate whether the information about bug types is encoded in the description. Particularly if *d* contains certain keywords about various types of bugs [17], such as *'mem', 'null', 'leak', 'overflow', 'injection', 'typo', 'bound', 'crash', 'unused'*, or *'uninit'*, then *d* provides the information about what issues *p* is fixing, the *what* aspect of *p*,

2. To determine whether the description contains information about how the patch is fixing bugs, we investigate whether the information about code modifications or actions is included. If *d* contains verbs or modification patterns describing the changes to fix the buggy version [18], such as *'add', 'fix', 'check', 'remove', 'delete', 'avoid', 'prevent', 'reject', 'replace'*, or *'Add [missing] ... [for|to...]'* then *d* provides the information about how *p* is fixing issues, the *how* aspect of *p*, and

3. If *d* starts with the name of functions/files/modules where the code modifications of *p* are made, then *d* provides the information about where *p* is fixing issues, the *where* aspect of *d*.

In our empirical study, we found that 11.0% of the descriptions contain all three aspects, while 57.6% of patch descriptions express 2/3 aspects, and about 89.0% of them convey at least one of the three aspects. Among the set of patches that convey at least one of the three aspects, we randomly sampled 385 cases with a confidence level of 95% and margin of error of 5% to investigate whether these patch descriptions contain information other than the *what*, *how*, and *where* aspects. After manually investigating the sampled cases, we found that only 7.8% of them contained other kinds of information, such as when the issue occurred. This implies that the majority of patch descriptions focus primarily on the essential aspects of *what*, *how*, and *where*, with a limited proportion including supplementary details.

Moreover, among 385 patches randomly sampled from the remaining 11% of the patches whose descriptions do not contain the aforementioned keywords, we observed that they still provide information about *what*, *how*, and *where* yet in different ways. For example, *"Reduce picture size for yadif"*.[2] More details about our empirical analysis can be found on our website [20].

As seen, the patch description in Fig. 1 is written following the convention, *"{where}: {what and how the patch fixes the bug}"*. This convention is *consistently* applied by the author (*Michael Niedermayer*) in describing more than 90% of his patches in FFmpeg. For example, two descriptions of the other patches created by the same developer in FFmpeg before the patch in Fig. 1 are *"avformat/avio: fix memory leak in url_find_protocol"* and *"avformat/avio: Fix unknown protocol handling"*. The recent study [21] has shown that syntactic and stylistic conventions of a particular project or user are important in describing code changes.

Overall, *the descriptions of patches provide descriptive information about **what**, **where**, and **how** the patch is fixing **in personalized or project-specific natural language***.

2.2. Key ideas

To enhance the effectiveness of patch description generation, our approach revolves around three core ideas: explicitly providing contextual information, enforcing syntactic and stylistic conventions through historical context, and framing patch description generation as a translation task.

1. Explicitly Providing Contextual Information for Patch Description Generation. Our first key idea focuses on the explicit provision of contextual information to convey essential elements in patch descriptions. Specifically, we aim to offer detailed insights into the *what*, *how*, and *where* aspects of a patch addressing a bug. This involves identifying and presenting information about the bug being addressed (*what*), detailing

the operations employed to fix the bug (*how*), and specifying the locations (e.g., functions, files, or modules) where modifications occur (*where*). By incorporating these aspects, our model learns to utilize such information effectively, enabling the generation of informative and context-rich descriptions for software patches.

2. Enforcing Syntactic and Stylistic Conventions Through Historical Context. The second key idea focuses on enforcing syntactic and stylistic conventions within a specific project or for an individual developer. By considering the historical context of patches, our model tailors its output to align seamlessly with established coding standards and preferences. This is achieved by incorporating recent patch descriptions as an input of our model, allowing the model to learn from the syntactic and stylistic conventions observed in the project's or developer's past patches. This approach ensures that the generated patch descriptions not only adhere to linguistic norms but also reflect individual or project-specific coding conventions, enhancing the integration of the generated patches into the existing codebase.

3. Patch Description Generation as Machine Translation with Dual-objective. Our third key idea is to frame patch description generation as a machine translation task with dual objectives. Besides the traditional objective in NMT, correctly generating expected descriptions, we also enforce NMT models to *explicitly* recognize and incorporate the similarities between descriptions. Particularly, the additional objective is for each patch, accurately predicting the affiliating group containing similar patches reflected by the descriptions. This third idea involves learning to express the *what*, *how*, and *where* aspects of a patch in personalized or project-specific natural language and to determine the group to which the patch belongs, facilitating a smoother comprehension of the introduced code changes.

3. PATCHEXPLAINER: Automated Description Generation for Software Patches

In this section, we introduce PATCHEXPLAINER, a novel automated description generation for software patches. In PATCHEXPLAINER, there are two main phases: *Contextual Information Extraction* and *Patch Description Generation*.

3.1. Contextual information extraction

A software patch is a structured representation of changes applied to a code repository. In this work, we formally define software patches as follows.

Definition 3.1. (Software Patch). Given a repository, a software patch *p* is a 5-tuple $p = \langle C, \Delta, a, t, d \rangle$ where:

- *C* is a buggy code version
- $\Delta = \{c_1, c_2, \ldots, c_n\}$ is a set of code changes. A change $c \in \Delta$ is a pair $c = \langle o, s \rangle$ where:
  - *o* is an operation change, $o \in \{added, deleted\}$
  - *s* is a changed statement
- *a* is the patch author
- *t* is the fixing time
- *d* is the patch description

3.1.1. Semantic context extraction

For a patch, code changes are introduced to fix a bug. In other words, the aspect of *how* the code is corrected is conveyed by the code changes. The changes impact the (existing) buggy code to correct the code's unexpected behaviors. Intuitively, the buggy version should be required to capture *what* bug in the code is fixed by the patch. However, not every part of the buggy code is semantically related to the bug and patch. To avoid redundancy in capturing the meaning of the patch, the code statements unrelated to the bug and patch should not be provided.

---

[2] https://github.com/FFmpeg/FFmpeg/commit/02d0803

For example, in Fig. 1, the code that is not related to the changed statements at lines 5 and 6 via program dependencies (i.e., control/data dependency), such as lines 7–9, should not be considered. In this work, we represent the information conveying the *what* and *how* aspects of a patch in a compact representation (*Patch-related Code*) containing the patch's code changes and their related (unchanged) code of the buggy code version.

Definition 3.2. (Patch-related Code). For a patch $p = \langle C, \Delta, a, t, d \rangle$, the patch-related code of p, R_p is the code sequence containing (annotated) changed statements in Δ and the unchanged statements in C which are **semantically** related to the changed statements. Specifically, the set of related unchanged statements, U, consists of statements in C and not in Δ, such that $s \in U$ if:

- $\exists c = \langle o, s' \rangle \in \Delta$, such that there exists a data/control dependency between statement s and statement s', or
- $\exists s'' \in U$, such there exists a data/control dependency between the statements s and s''.

Note that the statements in the patch-related code of p remain the same relative order in the buggy code version C because the order of the statements is crucial for determining the meaning of the code.

The *scope* of a patch provides information about *where* a specific code segment is changed, i.e., changed function or file. Thus, we will leverage the information related to the file and function that contains the changed code in a patch. In our empirical study, we found that about 73% of the patch descriptions contain the changed functions/files' names.

Definition 3.3. (Patch Scope). For a patch $p = \langle C, \Delta, a, t, d \rangle$, the patch scope of p, S_p, consists of the information about the file names and function names where p modifies in the buggy code C.

For example, the scope of the patch shown in Fig. 1 includes the name of the changed function, `url_find_protocol`, and the path of the modified file, *"libavformat/avio.c"*. In this example, the file path is used at the beginning of the patch description.

3.1.2. Syntactic context extraction

The syntactic and stylistic conventions in patches' descriptions help patches and their descriptions readability and searchability. These conventions refer to the rules that order/group the information about patches (including the *what*, *how*, and *where* aspects) by a certain style. However, the conventions of patches could be very different for each project and individual developers [21]. To aid syntactic and stylistic conventions of a particular project or developer in patch description generation, we consider their historical patch descriptions as a part of our generation model's input.

Definition 3.4. (Historical Descriptions). For a patch $p = \langle C, \Delta, a, t, d \rangle$, the set of historical descriptions of p, H_p, consists of the patch descriptions which are created by the same author a of p before fixing time t in the same project.

For instance, the description of the patch shown in Fig. 1 is *"avformat/avio: Fix null pointer dereference in case of memleak"*, while a few historical descriptions of the patch are {*"avformat/avio: fix memory leak in url_find_protocol"* and *"avformat/avio: Fix unknown protocol handling"*}. All of these descriptions start with the information in the scope of the corresponding patches.

In this work, we formulate the task of patch description generation as a translation task transferring a sentence, S_p in the source language to the corresponding sentence, $\mathcal{T}_p$ in the target language. In this translation task, S_p from the source language is the sequence constructed from a software patch $p = \langle C, \Delta, a, t, d \rangle$, $S_p = concat[R_p, concat(S_p), concat(H_p)]$, where *concat* is the concatenating function returning a sequence joining sub-sequences by separators; R_p, S_p, and H_p are the patch-related code, patch scope, and historical description set of p. Meanwhile, $\mathcal{T}_p$ from the target language is the description d of p.

3.2. Patch description generation

Our idea is to design a patch description generation model as a translation model with the awareness of the similarity between descriptions. The rationale is that enforcing the model learns to additionally the similarity between descriptions could help the model generate descriptions more accurately. The similarity between patch descriptions could be manifested in various forms to improve patch description generation, such as recurring bug-fix types, shared keywords or phrases, and similar severity levels of addressed issues. However, the challenge lies in the absence of explicit labels indicating such similarities between patches and their descriptions. To capture implicit/hidden similarities between descriptions, we introduce a crucial step in our approach: *Clustering patches based on description similarity* (Step 2 in Fig. 2). This step enables us to organize patches into groups based on shared characteristics and identify meaningful patterns within the varied range of patch descriptions.

Subsequently, our translation model is trained with a dual-objective: *accurately generating descriptions for individual patches* and *precisely predicting the cluster or group to which each patch belongs* (Step 3 in Fig. 2). This dual-task framework leverages the benefits of both semantic grouping and accurate description generation, enabling the model to capture the meaning of code changes while fostering coherence and consistency within identified clusters. Moreover, predicting cluster affiliation could be considered an easier "short-term" task than the harder "long-term" task, patch description generation. The existing studies [22] have shown that the model tends to perform better once it is trained with both the "short-term" and "long-term" objectives. The impact of the consideration similarity between patch descriptions on the description generation performance of this dual-task framework will be experimentally evaluated in Section 5.

3.2.1. Clustering patches based on description similarity

Our approach incorporates a crucial step, clustering patches based on the similarity of their descriptions. This process creates cohesive groups of patches with shared semantic characteristics in their language representations. By employing established algorithms [23], such as K-means or hierarchical clustering, we categorize patches into distinct groups where descriptions exhibit notable similarity.

For each patch $p = \langle C, \Delta, a, t, d \rangle$ in the training data, the description d is transformed into D-dimensional hidden features, v_p using embedding techniques [24], such as TF-IDF [25], Word2vec [26], or CodeBERT [27]. Next, the clustering process relies on standard algorithms [23] that assess the semantic similarity of patch descriptions. These algorithms use the embedded vectors corresponding to the patch descriptions to categorize patches into groups with similar descriptions. Formally, based on vector v_p, patch p is categorized into the group $\mathcal{G}_p$ containing the patches whose descriptions are semantically similar to d in this step. The primary goal is to capture detailed semantics in descriptions, identifying patches that address similar characteristics, such as types of issues, or relate to specific functionalities within the codebase. The grouped patches serve as enriched training data for subsequent phases, contributing to the creation of a robust translation model.

3.2.2. Training translation model with dual objectives

Following patch clustering based on description similarity, a translation model, $\mathcal{M}$, is applied to learn to *generate descriptions* and *predict cluster affiliation* from the provided contextual information. Particularly, for each patch $p = \langle C, \Delta, a, t, d \rangle$ in the training data, $\mathcal{M}$ is trained to learn generating the description ($\mathcal{T}_p = d$) and predicting cluster affiliation of p (the clustered group $\mathcal{G}_p$). $\mathcal{M}$ is trained with dual-objective to enhance the accuracy and coherence of generated descriptions.

Objective 1, Accurate Description Generation: The first objective involves training the model to generate descriptions for individual patches accurately. This task is central to ensuring that the model
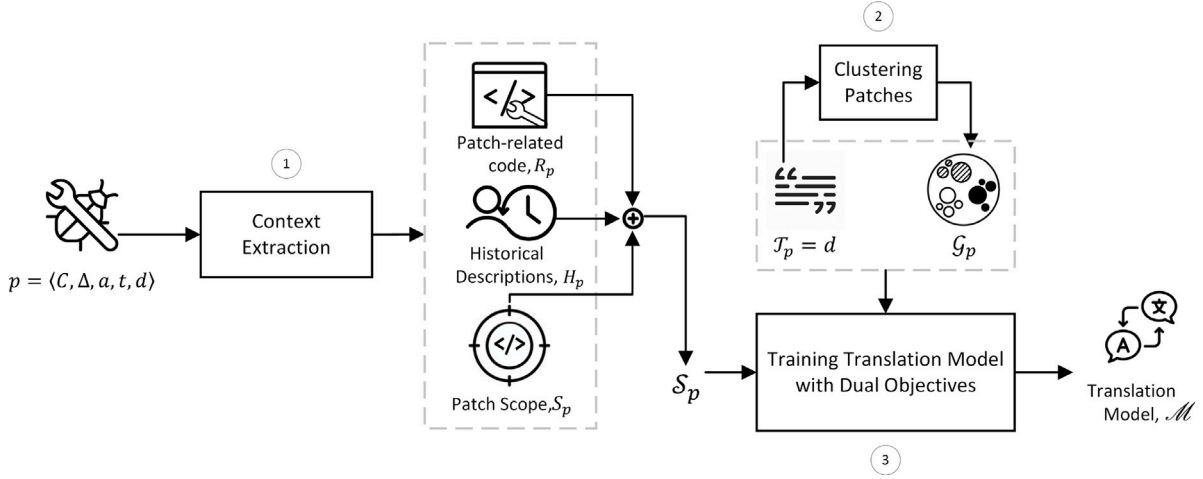
Fig. 2. PATCHEXPLAINER: An automated patch description generation approach.

captures the meaning of the provided contextual information, including *patch-related code, patch scope*, and *historical descriptions*, and communicates them effectively in natural language. Specifically, we compare the accuracy of the predicted translation ($\hat{\mathcal{T}}_p$) to the actual translation ($\mathcal{T}_p$) of the source sentence S_p to compute a loss. While there are several varieties of loss functions, we apply the following common version of the Cross-Entropy Loss:

$$\mathcal{L}_1 = -\sum_{w=1}^{|\hat{\mathcal{T}}_p|} \sum_{e=1}^{|V|} y_{w,e} \log(\hat{y}_{w,e}) \tag{1}$$

In Formulation (1), V is the list of vocabulary entries. Additionally, $y_{w,e} = 1$ if the vocabulary entry e is the corrected word; otherwise, $y_{w,e} = 0$. This means that if the wth word of $\mathcal{T}_p$ is the vocabulary entry e in V, then $y_{w,e} = 1$; otherwise, $y_{w,e} = 0$. Meanwhile, $\hat{y}_{w,e}$ is the predicted probability of vocabulary entry e on word wth. In essence, $\mathcal{L}_1$ sums over the negative log likelihoods that the model gives to the correct word at each position in $\hat{\mathcal{T}}_p$. Given that the negative log function returns 0 when the input is 1 and increases exponentially as the input approaches 0, the closer the probability that the model gives to the correct word at each point in $\hat{\mathcal{T}}_p$ is to 100%, the lower the loss.

Objective 2, Cluster Affiliation Prediction: Simultaneously, the second objective entails training the model to predict each patch's cluster or group affiliation. This additional task encourages the model to understand broader patterns and themes within the dataset, fostering coherence and consistency in the generated descriptions. For this objective, we compare the accuracy of the predicted cluster affiliation ($\hat{\mathcal{G}}_p$), given S_p, to the cluster affiliation ($\mathcal{G}_p$) determined by the patch clustering step (Step 2 in Fig. 2). Formally:

$$\mathcal{L}_2 = -\sum_{i=1}^{N} g_i \log(\hat{g}_i) \tag{2}$$

In Formulation (2), N is the number of groups/clusters. Additionally, $g_i = 1$ if the patch is clustered into the group ith; otherwise, $g_i = 0$. Meanwhile, $\hat{g}_i$ is the predicted probability that the patch in the group ith. This can be done by considering the group index as a special token in target sequences.

Dual-Task Framework: The dual-task framework leverages the benefits of both accurate description generation and cluster affiliation prediction. By optimizing the model for these two objectives, we aim to strike a balance between precision in individual descriptions and a broader understanding of semantic clusters. Thus, the overall loss that the training process has to optimize for each software patch in the training data is $\mathcal{L} = \frac{\mathcal{L}_1 + \mathcal{L}_2}{2}$.

Table 1
Dataset statistics.

Repo	#Patches	AVG. tokens in d	#Changed files	#Changed LOCs
Linux	7,877	7.36	8,570	42,190
Ffmpeg	5,092	7.30	5,314	24,708
Media-tree	3,571	7.32	3,921	19,146
Qemu	3,039	6.82	3,349	17,706
550 projects more...				
Total	30,308	7.33	33,092	163,277

4. Evaluation methodology

To evaluate our automated description generation approach for software patches, we seek to answer the following research questions:

RQ1: *Accuracy and Comparison*. How accurate is PATCHEXPLAINER in generating software patch descriptions? And how is it compared to the state-of-the-art approaches [10–13,28]?

RQ2: *Intrinsic Analysis*. How do different aspects of our approach impact the description generation performance of PATCHEXPLAINER, including clustering algorithms and the description generation model's properties/components?

RQ3: *Context Analysis*. How do provided contextual information, including patch scopes and historical descriptions impact PATCHEXPLAINER's performance?

RQ4: *Sensitivity Analysis*. How do various input's factors, including training data size and changed code's complexity, and bug types affect PATCHEXPLAINER's performance?

RQ5: *Time Complexity*. What is PATCHEXPLAINER's running time?

4.1. Dataset

To evaluate patch description generation performance, we construct a large dataset of 30 K software patches fixing bugs and vulnerabilities in 554 open-source projects (Table 1). This dataset is collected from other public datasets in the existing work [14,19] and National Vulnerability Database (NVD) [29].

To extract the patch descriptions, we applied the same procedure as the method applied by Mahbub et al. [14]. Particularly, the commit messages of these patching commits could be considered as their descriptions. We also followed the same data cleaning step of those studies to remove unique tokens (e.g., commit IDs, URLs, developer names, bug IDs, emails), which could be impossible to generate correctly. Additionally, we found that 96% of the collected patches have descriptions containing 3 to 15 tokens. Thus, we eliminated the cases where the descriptions were not representative, were too short (fewer than three tokens), or were too long (more than 15 tokens).

4.2. Evaluation setup, procedure, and metrics

4.2.1. Empirical procedure

RQ1. Accuracy and Comparison.

Baselines. We compared PATCHEXPLAINER against the state-of-the-art patch description generation approaches:

1. **NNGen** [10] is an IR-based commit message prediction technique.
2. **Coregen** [11] is a pure Transformer-based approach for representation learning targeting commit message generation.
3. **RACE** [12] is a context-aware retrieval-based deep commit message generation approach combining language modeling and retrieval advantages.
4. **COME** [13] combines retrieval techniques with translation-based methods through a decision algorithm, and this method learns better contextualized code change representation
5. **GPT-3.5 Turbo** [30] is a large language model (LLM) capable of generating different creative text formats, including descriptions of code changes in the task of patch description generation.

We used the implementation in their original papers for all the baseline approaches.

Data Splitting Strategies. In this comparative study, to assess the generalizability of the approaches, we evaluated the performance of the approaches in two real-world settings: *development-process* and *cross-project*:

Development- (dev.-) process setting. This strategy investigates how well the models perform on unseen data from a different time period. Particularly, we divided the patches into those before and after time point *t*. The patches before *t* were used for training, while the patches after *t* were used for evaluation. We selected a time point *t* to achieve a training/testing split ratio of 80/20 based on time. Specifically, for the *dev.-process* setting, the patches from Aug 1987 to Sep 2016 were used for training, and the patches from Oct 2016 to Jul 2023 were used for evaluation. In total, the training/testing split in the number of patches for this setting is 24,264/6,062.

Cross-project setting. We evaluated how well the approaches can learn to generate patch descriptions in one set of projects and generate descriptions for patches in the other. Specifically, the patches in a fixed set of projects were used to train the approaches, and the remaining patches were used for testing. For this setting, the whole set of projects was randomly split into 80% (455 projects) for training and 20% (107 projects) for testing. The training/testing split in the number of patches for this setting is 24,375/5,952.

These two splitting strategies allow us to evaluate the model's performance in different scenarios:

- *Dev.-process* setting (*time-aware* evaluation) assesses the model's ability to adapt to potential changes in coding styles or bug patterns over time.
- *Cross-project* setting assesses the model's ability to generalize to unseen projects, demonstrating its broader applicability.

Evaluation Setup. For a practical evaluation, we considered a maximum of ten statements in the patch-related code and the last ten historical descriptions of each patch. We used the *K*-mean clustering algorithm and Code Llama 7B [31] parameters fine-tuned using Lora [32] as the translation model in PATCHEXPLAINER. For detailed implementation and other hyper-parameters, one can see our website [20]. Note that all our experiments were run on a server with GeForce RTX 4090 GPU.

RQ2. Intrinsic Analysis. We also investigated the impact of different aspects of our approach on the description generation performance:

- Description generation model: We investigated the impact of the description generation model's properties on the performance of PATCHEXPLAINER, including the translation model and the additional objective (*Cluster Affiliation Prediction*).

- Patch clustering: We studied the impact of various aspects of the clustering step, including the clustering algorithm and embedding techniques on PATCHEXPLAINER's performance.

RQ3. Context Analysis. We investigated the impact of the provided contextual information on the description generation performance of PATCHEXPLAINER. Note that, the patch-related code is considered the essential part of patches. Thus, we evaluated the impact of providing path scope and historical descriptions as input of PATCHEXPLAINER.

RQ4. Sensitivity Analysis. We studied the impacts of the following factors on the performance of PATCHEXPLAINER: training size and change complexity. To systematically vary these factors, we gradually added more training data and varied the range of the change rate.

4.2.2. Metrics

To evaluate the patch descriptions generated by PATCHEXPLAINER, we used several metrics at both *surface* and *semantic* levels. In general, the surface-level metrics, including *BLEU*, *METEOR*, *ROUGE*, and *Exact Match*, focus on lexical and syntactic aspects of descriptions, while the semantic-level metrics delve into the meaning or semantics of the generated content. Collectively, these metrics provide a comprehensive evaluation of the quality and alignment of generated descriptions with reference descriptions in the patch description generation task.

(1) *Surface-Level Metrics*:

- **BLEU** (*Bi-Lingual Evaluation of Understanding*) measures how many word sequences from the actual description occur in the generated description and uses a *n*-gram precision to generate a score:

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

where p_n is the ratio between overlapping *n*-grams from both generated and actual descriptions, and the total number of *n*-grams in the generated description, and w_n is the weight of the *n*-gram length. Following the existing studies [10–13], we use $N = 4$ and $w_n = 0.25$ for all $n \in [1, N]$. The brevity penalty, *BP*, lowers *BLEU* if the generated one is too small.

- **METEOR** (*Metric for Evaluation of Translation with Explicit ORdering*) is an F-Score-Oriented metric for measuring the performance of translation models, $MET. = F\text{-weighted}_M(1-p)$, where *p* is the chunk penalty, $p = 0.5(c/u_m)^3$, with C is the number of chunks in the generated description, U_m is the number of unigrams in the candidate. The harmonic mean of unigram precision and recall, with recall weighted higher than precision, $P_M = m/w_t$, $R_M = m/w_r$, and $F\text{-weighted}_M = \frac{10P_M R_M}{R_M + 9P_M}$, where *m* is a number of unigrams in the generated description also found in the actual one, while w_t and w_r are the numbers of unigrams in generated one and actual one.

- **ROUGE** (*Recall-Oriented Understudy for Gisting Evaluation*) is one set of metrics for comparing the automatically generated descriptions against the actual descriptions. We focus on *ROUGE-L*, which computes the Longest Common Subsequence (*LCS*). Particularly, *ROUGE-L* recall:

$$R_{RL} = \frac{num\_tokens(LCS(\mathcal{T}, \hat{\mathcal{T}}))}{k}$$

and *ROUGE-L* precision:

$$P_{RL} = \frac{num\_tokens(LCS(\mathcal{T}, \hat{\mathcal{T}}))}{t}$$

where *k* and *t* are the numbers of tokens in $\mathcal{T}$ and $\hat{\mathcal{T}}$; *ROUGE-L* F1-score $F\text{-}1_{RL} = \frac{2P_{RL} R_{RL}}{R_{RL} + P_{RL}}$.

- **Exact Match** measures the rate of the cases where the generated description exactly matches the corresponding actual description. It is analogous to string equality checks in many programming languages, which are case-sensitive and space-sensitive.

Table 2
Cross-project: Patch description generation performance.

Method	BLEU	MET.	ROU.	EM	SemSim
NNgen	9.8	6.65	8.13	0.49	25.02
CoreGen	11.12	7.93	13.51	0.02	33.29
GPT-3.5 Turbo	13.50	13.50	12.85	0.07	50.30
RACE	18.23	15.61	19.26	0.29	51.69
COME	18.56	16.66	19.44	0.57	53.25
PATCHEXPLAINER	**23.94**	**20.53**	**27.06**	**1.26**	**58.69**

Table 3
Dev.-process: Patch description generation performance.

	BLEU	MET.	ROU.	EM	SemSim
NNgen	8.98	6.03	6.92	0.00	23.93
CoreGen	10.86	7.21	11.99	0.05	30.02
GPT-3.5 Turbo	13.92	13.86	13.68	0.02	50.94
RACE	18.48	15.53	18.21	0.00	49.10
COME	18.20	16.09	18.82	0.16	52.39
PATCHEXPLAINER	**25.98**	**22.01**	**28.99**	**1.40**	**60.78**

(2) *Semantic-level Metric*: In this work, we applied a similar semantic-level metric utilizing deep-pretrained word embedding techniques to measure the semantic similarity between the generated description and the actual one. Following the existing studies [14,33], we apply Sentence-BERT [34], trained on a large amount of code and text to provide a fixed-length numeric representation for any given description. We compute the *Semantic Similarity* as follows:

$$SemSim(\mathcal{T}_p, \hat{\mathcal{T}}_p) = cosine(sbert(\mathcal{T}_p), sbert(\hat{\mathcal{T}}_p))$$

where $sbert(x)$ is the numerical representation from Sentence-BERT for any input text x, $\mathcal{T}_p$ is the actual description, and $\hat{\mathcal{T}}_p$ is the generated description.

Note that, for every considering metric M, a description generation approach with higher M is better.

5. Experimental results

5.1. Answering RQ1: Accuracy and comparison

Tables 2 and 3 show the performance of PATCHEXPLAINER and the existing approaches, including NNgen [10], CoreGen [11], RACE [12], COME [13], and GPT-3.5 Turbo in the *cross-project* and *dev.-process* settings. As seen, PATCHEXPLAINER significantly outperforms all the state-of-the-art approaches in all considering metrics in both settings.

For both settings, in terms of *surface-level* similarity, PATCHEXPLAINER demonstrates substantial performance enhancements when compared to COME. For instance, the relative improvements of PATCHEXPLAINER compared to COME in the *dev.-process* setting are significant: **43%** in *BLEU*, **37%** in *METEOR*, and **54%** in *ROUGE-L F1*. Meanwhile, PATCHEXPLAINER's performance is **over three to four folds** better than those of NNgen, CoreGen, and RACE across all these metrics. PATCHEXPLAINER consistently outperforms the existing approaches, these signify a significant advancement in linguistic precision. Especially, PATCHEXPLAINER's *Exact Match* rate surpasses those of NNgen, CoreGen, RACE, and COME, reflecting its ability to generate descriptions that precisely match the reference descriptions. In terms of *semantic-level* similarity, PATCHEXPLAINER consistently raises the bar, demonstrating improvements ranging from **10%** to an impressive **154%** when compared to the existing approaches. This indicates a substantial advancement in capturing the underlying meaning or intent of code changes.

Moreover, while the performances of the other approaches remain stable between the two settings, PATCHEXPLAINER's performance could improve by 9% in the *dev.-process* setting compared to its performance in the *cross-project* setting. The reason for this improvement could be that PATCHEXPLAINER could effectively learn and utilize the vocabulary in the same project's past.

```
File path: libarchive/archive_read_support_format_zip.c
Git difff:
@@ -905,6 +905,7 @@ zip_read_local_file_header(struct archive_read *a, struct
archive_entry *entry,
            archive_wstrcat(&s, wp);
            archive_wstrappend_wchar(&s, L'/');
            archive_entry_copy_pathname_w(entry, s.s);
+           archive_wstring_free(&s);
        }
    } else {
        cp = archive_entry_pathname(entry);
@@ -915,6 +916,7 @@ zip_read_local_file_header(struct archive_read *a, struct
archive_entry *entry,
        archive_strcat(&s, cp);
        archive_strappend_char(&s, '/');
        archive_entry_set_pathname(entry, s.s);
+       archive_string_free(&s);
    }
  }
 }
```

Human-generated:
  Plug memory leak in zip_read_local_file_header()
PATCHEXPLAINER:
  Fix memory leak in zip_read_local_file_header()
COME:
  Fix memory leak
RACE:
  Fix memory leak
CoreGen:
  Sparc64: Like x86 we should check current->mm during perf backtrace generation.
NNGen:
  Free slab on corruption

Fig. 3. The descriptions generated by PATCHEXPLAINER and the others for patch 8fd7839 in `libarchive`.

Overall, *the consistent and significant improvements across these metrics demonstrate PATCHEXPLAINER's effectiveness in not only achieving high linguistic precision but also in conveying richer semantic content, demonstrating the high potential of our technique in generating patch descriptions*.

Result Analysis. Analyzing the results, we found that the main reason for the higher performance of PATCHEXPLAINER compared to the existing approaches is that PATCHEXPLAINER provides explicit context to generate patch descriptions. Moreover, by training the translation model with dual-objective, the generated descriptions contain precise and comprehensive semantics in multiple aspects. For example, Fig. 3 shows a patch in `libarchive`.[3] The description generated by PATCHEXPLAINER correctly contains the *what* (‘‘`memory leak`’’) and *where* (function ‘‘`zip_read_local_file_header`’’) which match the information in the ground truth. On the other hand, the descriptions generated by COME and RACE can only provide the bug type fixed by the patch, while the descriptions generated by CoreGen and NNGen are completely irrelevant.

Among the cases where PATCHEXPLAINER performed poorly, our observation is that a large portion of them have long descriptions. This is expected because it is harder to generate longer descriptions. Indeed, as shown in Fig. 4, the performance of PATCHEXPLAINER decreases gracefully when generating longer descriptions. Fig. 5 shows low-quality descriptions generated by PATCHEXPLAINER and the other approaches. In this example, even though the description generated by PATCHEXPLAINER is quite different from the expected one, it still correctly expresses information about the changed file and operational fixing details. In contrast, the other techniques fail to deliver accurate information about the patch.

Interestingly, we found several cases where the descriptions were not well generated by PATCHEXPLAINER regarding the considered metrics. Yet, they were even more informative than the human-generated ones. For example, the patch in Fig. 6 addresses mishandling ‘‘`.`’’. directory traversal in a mailbox name in function `imap_hcache_open`.[4] The description written by the developer for this patch is not very
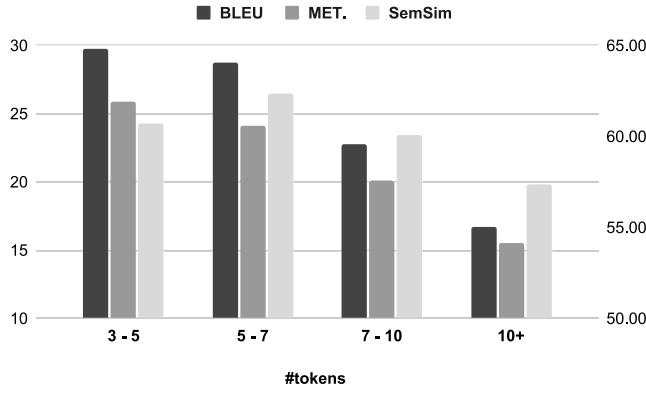

Fig. 4. Performance of PATCHEXPLAINER in different description complexity levels (left axis: *BLEU* & *MET.*; right axis: *SemSim*).
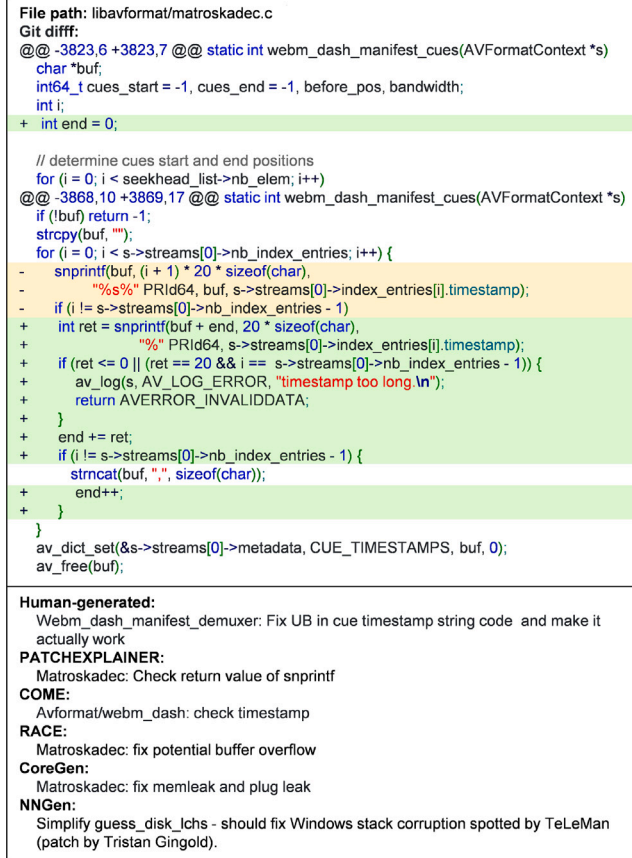

Fig. 5. Patch `8e6b9ef` in FFmpeg and the descriptions.
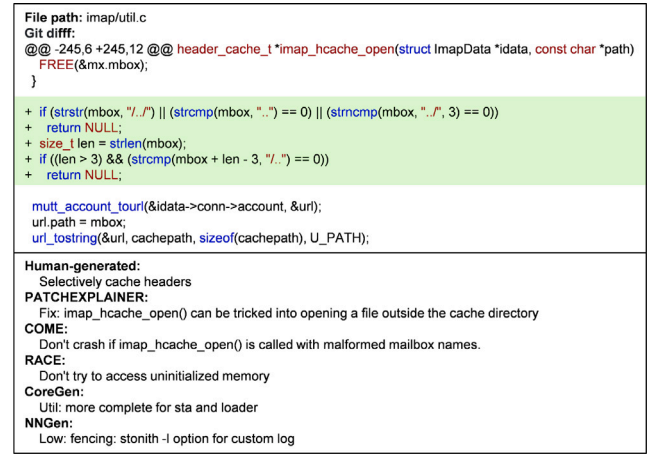

Fig. 6. An interesting generated patch description by PATCHEXPLAINER (CVE-2018-14355).

Table 4
Performance comparison on MCMD dataset [35] (C/C++).

	BLEU	MET.	ROU.
NNgen	13.61	9.39	18.21
CoreGen	16.74	11.72	22.83
RACE	19.13	12.55	24.52
COME	20.80	14.55	27.01
PATCHEXPLAINER	22.88	17.03	32.66

Table 5
Performance of PATCHEXPLAINER by translation model.

	BLEU	*MET.*	*ROU.*	*EM*	*SemSim*
Transformer	11.86	7.62	9.57	0.08	29.45
CodeT5	24.23	21.93	25.78	1.17	59.29
Code Llama	25.98	22.01	28.99	1.40	60.78

5.2. Answering RQ2: Intrinsic analysis

5.2.1. Impact of translation model

To evaluate the impact of translation models on the performance of PATCHEXPLAINER, we compare the performance of PATCHEXPLAINER with three representative translation models: Transformer [36], CodeT5 [37], and Code Llama 7B [31] in the *dev.-process* setting. In this experiment, we use all of the contexts of PATCHEXPLAINER; we use the *K*-means algorithm with *Tf–idf* embedding to cluster description and apply it to the *dev.-process* setting. The performance of PATCHEXPLAINER with different translation models is shown in Table 5. As expected, PATCHEXPLAINER achieves the best performance with Code Llama 7B, which can capture the meaning of code and text [31]. Meanwhile, PATCHEXPLAINER with CodeT5 with 220M parameters shows quite competitive results, with a margin of 7% and a 119% improvement in BLEU compared to PATCHEXPLAINER with Code Llama and traditional Transformer (110M parameters). The reason is that CodeT5 and Code Llama are larger models and pre-trained with much data. However, CodeT5 and Code LLama cost 2–10 times in training and inferring. Thus, the transformer could be suitable to be applied in PATCHEXPLAINER when deploying on machines with limited resources. At the same time, CodeT5 and Code Llama could be suitable for the cases preferring the patch description generation effectiveness.

5.2.2. Impact of training with dual-objective

Table 6 compares the performance of PATCHEXPLAINER under different training variants: Single-Objective and Dual-Objective. The results in Table 6 demonstrate that the Multi-Objective training variant outperforms the Single-Objective variant across all metrics. Particularly, the Multi-Objective training yields an improvement of 6.3% and 3.5%

informative. COME can generate an acceptable description for this case. Meanwhile, the description generated by PATCHEXPLAINER correctly describes the bug fixed by the patch and the fixed function.

To further validate PATCHEXPLAINER's generalizability, we conducted an additional evaluation on the MCMD dataset (C/C++) [35], which is commonly used to benchmark commit message generation approaches [12,13]. As shown in Table 4. PATCHEXPLAINER achieves significant improvements across all evaluated metrics, with the improvements of 10%–68% in *BLEU*, 17%–81% in *METEOR*, and 20%–79% in *ROUGE-L*. These results on the MCMD dataset further support the effectiveness of PATCHEXPLAINER.

Table 6
Performance of PATCHEXPLAINER by training variants.

	BLEU	MET.	ROU.	EM	SemSim
Single-Objective	24.45	20.49	27.52	0.79	58.75
Dual-Objective	25.98	22.01	28.99	1.40	60.78

Table 7
Performance of PATCHEXPLAINER by clustering algorithms.

	BLEU	MET.	ROU.	EM	SemSim
DBScan	24.32	21.96	25.87	1.25	59.52
K-mean	24.23	21.93	25.78	0.98	59.29
Gaussian Mixture	24.15	21.83	25.66	1.22	59.24
Agglomerative	24.60	22.15	26.04	1.47	59.52

Table 8
Performance of PATCHEXPLAINER by embedding methods.

	BLEU	MET.	ROU.	EM	SemSim
Tf-idf	24.23	21.93	25.78	0.98	59.29
Word2vec	24.43	22.01	25.85	1.35	59.40
CodeBERT	24.34	21.90	25.71	1.29	59.45

Table 9
Impact of patch scope on PATCHEXPLAINER's performance.

	BLEU	MET.	ROU.	EM	SemSim
W/O Patch Scope	18.26	16.86	20.45	0.36	50.67
W/ Patch Scope	24.23	21.93	25.78	0.98	59.29

Fig. 7. Impact of historical descriptions (left axis: *BLEU* & *ROU.*; right axis: *SemSim*).

in *BLEU* and *Semantic Similarity*, respectively. Especially, the Exact Match rate of PATCHEXPLAINER with Dual-Objective is about 80% higher than that of PATCHEXPLAINER with Single-Objective. These results suggest that incorporating the dual objectives of maximizing similarity between generated and actual descriptions and accurately predicting affiliating groups significantly enhances the overall performance of PATCHEXPLAINER. Therefore, the dual-objective training approach is recommended for optimal automated patch description generation results.

5.2.3. Impact of clustering algorithm

In Table 7, we present the evaluation of PATCHEXPLAINER's performance using different representative clustering algorithms [23]. The considered algorithms include K-means (Centroid-based), DBScan (Density-based), Gaussian Mixture (Distribution-based), and Agglomerative (Hierarchical), each configured with an identical number of clusters. The results show the stability of PATCHEXPLAINER across diverse clustering algorithms, showcasing consistent performance with maximum variations of 1.9% and 0.5% in *BLEU* and *Semantic Similarity*, respectively. This robust performance shows that PATCHEXPLAINER's effectiveness is maintained regardless of the clustering algorithm employed. Thus, the choice of clustering algorithm should lean toward simplicity and efficiency, making lightweight algorithms like K-means more preferable over sophisticated alternatives such as Gaussian Mixture.

We further investigated the influence of different embedding methods on the clustering performance of PATCHEXPLAINER, particularly when using the K-means algorithm. The considered embedding techniques include Tf-idf [25], Word2Vec for non-contextual embedding [26], and CodeBERT for contextual embedding [27]—all representative methods from prior work [24]. Table 8 illustrates the stability of PATCHEXPLAINER's performance across various embedding methodologies during the clustering step. Notably, PATCHEXPLAINER showcases consistent and reliable results regardless of the chosen embedding method. This indicates that PATCHEXPLAINER's efficacy in generating patch descriptions remains robust across different ways of representing input sequences. Among the evaluated embedding methods, even the traditional and interpretable Tf-idf approach proves to be a viable choice for PATCHEXPLAINER. This observation suggests that simplicity and interpretability, as offered by Tf-idf, can be favored without compromising the effectiveness of PATCHEXPLAINER in the context of patch description generation.

In general, with the common objective of grouping similar patches of the studied clustering algorithms, the groups resulting from the different algorithms and embedding techniques still contain the similar patches within them. Thus, PATCHEXPLAINER with dual-objectives training can effectively recognize and incorporate the similarities between patches in generating descriptions.

5.3. Answering RQ3: Context analysis

5.3.1. Impact of patch scope

To study the impact of considering the patch scope on PATCHEXPLAINER's performance, we compare the performance of PATCHEXPLAINER

using the inputs with and without the patch scopes. In this experiment, we use CodeT5 as a translation model, using the K-means algorithm with *Tf–idf* embedding to cluster descriptions. As shown in Table 9, the performance of PATCHEXPLAINER increases by 33% in *BLEU* and 17% in *Semantic Similarity* when considering the patch scope as a part of the input for generating description. This result aligns well with our empirical about the *where* aspect of patch descriptions in practice and confirms our method considering patch scopes as a part of models' input.

5.3.2. Impact of historical descriptions

To investigate the impact of historical descriptions on PATCHEXPLAINER performance, we gradually increase the number of historical descriptions from 0 to 10. In this experiment, we use CodeT5 as a translation model, using the K-means algorithm with *Tf–idf* embedding to cluster description and apply it to the *dev.-process* setting. As shown in Fig. 7, PATCHEXPLAINER can produce much better patch descriptions, up to about 18%, when considering historical descriptions. This is because historical description helps the model capture a specific style or personalize convention of the author. Thus, the descriptions generated between PATCHEXPLAINER and the developer may be in the same style and expression. In addition, when increasing the number of historical descriptions from 1–10, the performance of the PATCHEXPLAINER is slightly improved from 23.18–24.23 in BLEU. These results indicate that although historical descriptions could be valuable for generating patch descriptions in PATCHEXPLAINER, considering more historical descriptions could not provide more new valuable information for generating patch descriptions. Thus, the list of historical descriptions should be shortened to maintain both the effectiveness and efficiency of PATCHEXPLAINER.

5.4. Answering RQ4: Sensitivity analysis

5.4.1. Impact of training dataset

To measure the impact of training data size on PATCHEXPLAINER performance, we gradually increased the sizes of the training dataset by expanding the period for training data. As shown in Fig. 8, PATCHEXPLAINER demonstrates impressive efficiency in learning from training
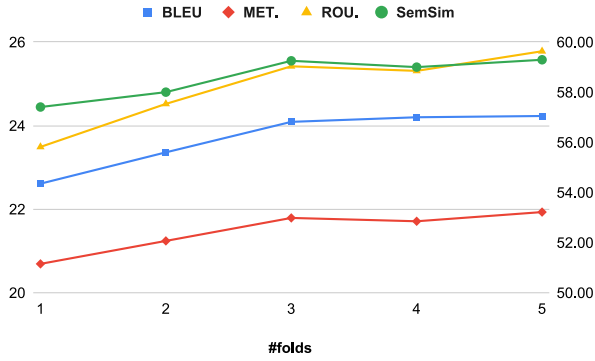
Fig. 8. Impact of training data size (left axis: *BLEU* & *MET.*, and *ROU.*; right axis: *SemSim*).
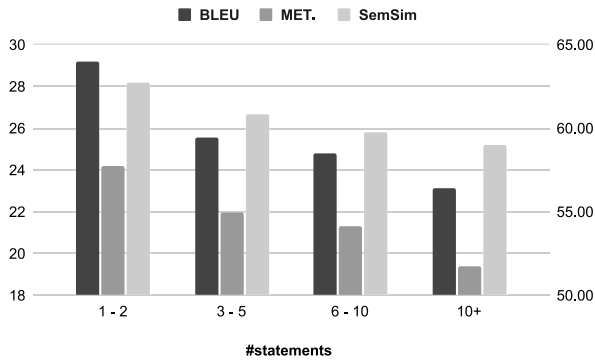

Fig. 9. Performance of PatchExplainer in different change complexity levels by #changed statements (left axis: *BLEU* & *METEOR*; right axis: *SemSim*).

data. Although with more data, PatchExplainer can learn to understand patches and translate descriptions better, PatchExplainer attains 93% of its full performance with only one-fifth of the training data (4.3 K out of 21.8 K instances). Moreover, PatchExplainer achieves 99% of its performance with 60% of the training data, significantly reducing the required training time. These results show PatchExplainer's capability to efficiently leverage a relatively smaller portion of the training data while still achieving high-performance levels. Such efficiency is crucial when computational resources or time constraints necessitate optimizing training efficiency.

5.4.2. Impact of change complexity

To investigate the impact of change complexity on PatchExplainer's performance, we conducted evaluations using varying numbers of changed statements in the *dev.-process* setting. As shown in Fig. 9, the performance of PatchExplainer slightly decreases by 6% in *Semantic Similarity* when increasing the number of changed statements. This decline is anticipated because it is more challenging for PatchExplainer to capture the meaning of the patch having many changed statements. However, the slight decrease implies that PatchExplainer remains a reliable choice for generating patch descriptions even in the patches involving more changes.

5.5. Answering RQ5: Time complexity

In this work, all experiments were conducted on a GeForce RTX 4090 GPU with 24 GB RAM. PatchExplainer required about 1.0 s to extract the contextual information for each case. Additionally, the patch clustering step in PatchExplainer costed 45 s to group more than 25 K patches in the training dataset based on their description similarity. After that, PatchExplainer took approximately 10 h for fine-tuning Code Llama 7B [31] as its translation model. On average, PatchExplainer required totally 0.59 s to generate a patch description.

5.6. Threats to validity

The main threats to the validity of our work consist of internal, construct, and external threats.

Threats to internal validity include the hyperparameter settings of all of the baseline models as well as the ones chosen for PatchExplainer. To reduce this threat, we systematically varied the setting of PatchExplainer to study its performance (Section 5) and reused the implementation with the same setting in the papers of the baseline approaches [10–13]. A threat may come from the method used to extract patch-related code. To reduce this threat, we use Joern [38] code analyzer, which is widely used in existing studies [39,40]. Another threat mainly lies in the correctness of the implementation of our approach. To reduce such a threat, we carefully reviewed our code and made it public [20] so that other researchers could double-check and reproduce our experiments.

Threats to construct validity relate to the suitability of our evaluation procedure. We used *BLEU*, *METEOR*, *ROUGE*, and *Semantic Similarity*. They are the widely-used evaluation measures for in the studies in code/change-text translation [6–14]. Another potential threat is the lack of human evaluation, which could provide a more comprehensive assessment of our approach. To address this, we plan to incorporate a human evaluation method in future work, where experienced software developers will review and rate the generated descriptions based on criteria such as accuracy, completeness, and usefulness. Additionally, evaluation in controlled environments may pose a threat. To mitigate this, we evaluated the approaches in two settings (*dev. process* and *cross-project*) and investigated the performance in various scenarios and settings.

Threats to external validity mainly lie in the selection of translation models used in our experiments. To mitigate this threat, we select the representative models that are well-known for NLP and SE tasks [31,36,37]. Another threat could be the potential limitations of using a set of patches and a limited set of keywords in investigating the "what", "how" and "why" aspects in Section 2.1. To mitigate this threat, we select a large set of 24.2 K patches and perform a manual analysis of randomly sampled descriptions with multiple reviewers. We will expand the dataset and alternative set of keywords to further mitigate this threat. Moreover, our experiments are conducted on only the code changes of C/C++ projects. Thus, the results could not be generalized for other programming languages. In our future work, we plan to conduct more experiments to validate our results in other languages.

6. Related work

PatchExplainer closely relates to the work on **automated commit message generation** [10–13,21,28,41–45]. NNGen [10], an IR-based approach, leverages cosine similarity to identify the k most similar commit diffs (represented using CC2Vec [46]) from a pre-existing dataset, eliminating the need for a separate training phase. Coregen [11] adopts a purely Transformer-based architecture for learning representations specifically tailored to commit message generation. RACE [12] integrates context-awareness into a retrieval-based deep learning approach for generating commit messages, effectively combining the strengths of language modeling and information retrieval techniques. COME [13] employs a decision algorithm to combine retrieval strategies with translation-based methods, enabling the model to learn more contextually-aware representations of code changes. While these approaches can generate commit messages, they often focus on explaining general code changes rather than the specific bug fixes addressed by the patch. These approaches implicitly learn semantic and conventional information during description generation. In contrast, our method is specialized for patch descriptions by explicitly providing semantic and conventional information, achieving demonstrably superior performance in Section 5.

Our work aligns with the broader area of **code-to-text translation**, where existing approaches translate code into natural language for various purposes like generating code comments, bug explanations, and source code summaries. Recent advancements heavily rely on learning-based methods [7–9,47–54]. Notably, Wei et al. [50] combine both IR and NMT in comment generation. Mastropaolo et al. [51] leverage the powerful Text-To-Text Transfer Transformer (T5) for various tasks including code comments. Geng et al. [53] leverage large language models (LLMs) for code comment generation, addressing the limitation of traditional methods that generate only one comment per code snippet, which often fails to meet developers' diverse needs. Gao et al. [52] introduce a structure-guided transformer for code summarization, integrating code structural features into the Transformer architecture. This involves embedding local symbolic details like code tokens and statements, along with global syntactic structures such as data flow graphs, into the self-attention module of the Transformer. Xu et al. [54] propose UniLog which is an automatic logging framework leveraging the in-context learning paradigm of large language models (LLMs). Parvez et al. [14] propose Bugsplainer, a transformer-based generative model that generates natural language explanations for software bugs. Bugsplainer leverages structural information and buggy patterns from the source code to generate an explanation for a bug. Importantly, these approaches describe the source code itself, while PATCHEXPLAINER focuses on the specific changes introduced in the code, particularly the bug fixes. This complementary nature suggests that combining these approaches (including PATCHEXPLAINER) could provide developers with even more comprehensive explanations in natural language for various steps during the software development process.

Several **learning-based approaches** have been proposed for specific SE tasks such as code suggestion [7,55,56], code generation [57, 58], code translation [59–61], test generation [62,63], bug detection [64–67], bug fix identification [68], and program repair [69, 70].

7. Conclusion

In conclusion, our proposed approach, PATCHEXPLAINER, addresses the challenges associated with manual patch description generation by framing it as a machine translation task. Leveraging explicit representations, historical context, and syntactic conventions, PATCHEXPLAINER demonstrates consistent superiority over existing methods in experimental evaluations. With improvements of up to 189% in BLEU, 5.7X in Exact Match rate, and 154% in Semantic Similarity, PATCHEXPLAINER showcases its efficacy in automating the generation of software patch descriptions. The dual objectives of maximizing similarity and accurately predicting affiliating groups contribute to the robust performance of PATCHEXPLAINER, establishing it as a valuable tool in the software development cycle.

CRediT authorship contribution statement

Thanh Trong Vu: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Tuan-Dung Bui:** Software, Methodology, Investigation, Data curation. **Thanh-Dat Do:** Software, Data curation. **Thu-Trang Nguyen:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Resources, Project administration, Methodology, Investigation, Formal analysis. **Hieu Dinh Vo:** Writing – review & editing, Validation, Supervision, Resources, Project administration, Conceptualization. **Son Nguyen:** Writing – review & editing, Writing – original draft, Supervision, Project administration, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgement

References

[1] K. Liu, D. Kim, A. Koyuncu, L. Li, T.F. Bissyandé, Y. Le Traon, A closer look at real-world patches, in: 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2018, pp. 275–286.

[2] M. Monperrus, A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 234–242.

[3] J. Dadzie, Understanding software patching: Developing and deploying patches is an increasingly important part of the software development process, Queue 3 (2) (2005) 24–30.

[4] S. Reis, R. Abreu, H. Erdogmus, C. Păsăreanu, SECOM: Towards a convention for security commit messages, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 764–765.

[5] Y. Tian, Y. Zhang, K.-J. Stol, L. Jiang, H. Liu, What makes a good commit message? in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 2389–2401.

[6] S. Gao, C. Gao, Y. He, J. Zeng, L. Nie, X. Xia, M. Lyu, Code structure–guided transformer for source code summarization, ACM Trans. Softw. Eng. Methodol. 32 (1) (2023) 1–32.

[7] S. Nguyen, H. Phan, T. Le, T.N. Nguyen, Suggesting natural method names to check name consistencies, in: 2020 IEEE 42nd International Conference on Software Engineering, IEEE, 2020, pp. 1372–1384.

[8] X. Hu, G. Li, X. Xia, D. Lo, Z. Jin, Deep code comment generation, in: Proceedings of the 26th Conference on Program Comprehension, 2018, pp. 200–210.

[9] Z. Ding, H. Li, W. Shang, Logentext: Automatically generating logging texts using neural machine translation, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2022, pp. 349–360.

[10] Z. Liu, X. Xia, A.E. Hassan, D. Lo, Z. Xing, X. Wang, Neural-machine-translation-based commit message generation: how far are we? in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 373–384.

[11] L.Y. Nie, C. Gao, Z. Zhong, W. Lam, Y. Liu, Z. Xu, Coregen: Contextualized code representation learning for commit message generation, Neurocomputing 459 (2021) 97–107.

[12] E. Shi, Y. Wang, W. Tao, L. Du, H. Zhang, S. Han, D. Zhang, H. Sun, RACE: Retrieval-augmented commit message generation, in: Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, 2022, pp. 5520–5530.

[13] Y. He, L. Wang, K. Wang, Y. Zhang, H. Zhang, Z. Li, COME: Commit message generation with modification embedding, in: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023, pp. 792–803.

[14] P. Mahbub, O. Shuvo, M.M. Rahman, Explaining software bugs leveraging code structures in neural machine translation, in: 2023 IEEE/ACM 45th International Conference on Software Engineering, ICSE, IEEE, 2023, pp. 640–652.

[15] D. Bahdanau, K.H. Cho, Y. Bengio, Neural machine translation by jointly learning to align and translate, in: 3rd International Conference on Learning Representations, ICLR 2015, 2015.

[16] Y. Wu, M. Schuster, Z. Chen, Q.V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al., Google's neural machine translation system: Bridging the gap between human and machine translation, 2016, arXiv preprint arXiv:1609.08144.

[17] H.D. Vo, S. Nguyen, Can an old fashioned feature extraction and a lightweight model improve vulnerability type identification performance? Inf. Softw. Technol. (2023) 107304.

[18] J. Dong, Y. Lou, D. Hao, L. Tan, Revisiting learning-based commit message generation, in: 2023 IEEE/ACM 45th International Conference on Software Engineering, ICSE, IEEE, 2023, pp. 794–805.

[19] X. Wang, S. Wang, P. Feng, K. Sun, S. Jajodia, Patchdb: A large-scale security patch dataset, in: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN, 2021, pp. 149–160, http://dx.doi.org/10.1109/DSN48987.2021.00030.

[20] PatchExplainer, URL https://github.com/VNU-ISE/Patch-Description-Generation.

[21] A. Eliseeva, Y. Sokolov, E. Bogomolov, Y. Golubev, D. Dig, T. Bryksin, From commit message generation to history-aware commit message completion, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2023, pp. 723–735.

[22] T. Standley, A. Zamir, D. Chen, L. Guibas, J. Malik, S. Savarese, Which tasks should be learned together in multi-task learning? in: International Conference on Machine Learning, PMLR, 2020, pp. 9120–9132.

[23] D. Xu, Y. Tian, A comprehensive survey of clustering algorithms, Ann. Data Sci. 2 (2015) 165–193.

[24] Z. Ding, H. Li, W. Shang, T.-H.P. Chen, Can pre-trained code embeddings improve model performance? Revisiting the use of code embeddings in software engineering tasks, Empir. Softw. Eng. 27 (3) (2022) 1–38.

[25] P. Soucy, G.W. Mineau, Beyond TFIDF weighting for text categorization in the vector space model, in: Proceedings of the 19th International Joint Conference on Artificial Intelligence, 2005, pp. 1130–1135.

[26] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, in: Y. Bengio, Y. LeCun (Eds.), 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings, 2013.

[27] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou, CodeBERT: A pre-trained model for programming and natural languages, in: Findings of the Association for Computational Linguistics, EMNLP 2020, Association for Computational Linguistics, Online, 2020, pp. 1536–1547.

[28] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, J. Lu, Commit message generation for source code changes, in: IJCAI, 2019.

[29] PatchExplainer, URL https://github.com/VNU-ISE/Patch-Description-Generation.

[30] OpenAI, GPT-3.5 Turbo, 2024, https://platform.openai.com/docs/models/gpt-3-5-turbo. (Accessed 05 2024).

[31] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X.E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, et al., Code llama: Open foundation models for code, 2023, arXiv preprint arXiv:2308.12950.

[32] E.J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen, Lora: Low-rank adaptation of large language models, 2021, arXiv preprint arXiv:2106.09685.

[33] S. Haque, Z. Eberhart, A. Bansal, C. McMillan, Semantic similarity metrics for evaluating source code summarization, in: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, 2022, pp. 36–47.

[34] N. Reimers, I. Gurevych, Sentence-BERT: Sentence embeddings using siamese BERT-networks, in: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP, 2019, pp. 3982–3992.

[35] W. Tao, Y. Wang, E. Shi, L. Du, S. Han, H. Zhang, D. Zhang, W. Zhang, On the evaluation of commit message generation models: An experimental study, in: 2021 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2021, pp. 126–136.

[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, in: Advances in Neural Information Processing Systems, vol. 30, 2017.

[37] Y. Wang, W. Wang, S. Joty, S.C. Hoi, CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, in: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 8696–8708.

[38] F. Yamaguchi, N. Golde, D. Arp, K. Rieck, Modeling and discovering vulnerabilities with code property graphs, in: 2014 IEEE Symposium on Security and Privacy, IEEE, 2014, pp. 590–604.

[39] Y. Ding, S. Suneja, Y. Zheng, J. Laredo, A. Morari, G. Kaiser, B. Ray, VELVET: A novel ensemble learning approach to automatically locate VulnErable statements, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering, IEEE, 2022, pp. 959–970.

[40] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, VulDeePecker: A deep learning-based system for vulnerability detection, in: 25th Annual Network and Distributed System Security Symposium, The Internet Society, 2018.

[41] J. Dong, Y. Lou, Q. Zhu, Z. Sun, Z. Li, W. Zhang, D. Hao, FIRA: Fine-grained graph-based code change representation for automated commit message generation, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 970–981.

[42] T.T. Vu, T.-D. Do, H.D. Vo, Context-encoded code change representation for automated commit message generation, 2023, arXiv preprint arXiv:2306.14418.

[43] W. Tao, Y. Zhou, Y. Wang, H. Zhang, H. Wang, W. Zhang, KADEL: Knowledge-aware denoising learning for commit message generation, ACM Trans. Softw. Eng. Methodol. 33 (5) (2024).

[44] Y. Wu, Y. Li, S. Yu, Commit message generation via ChatGPT: How far are we? in: Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering, 2024, pp. 124–129.

[45] Y. Zhang, Z. Qiu, K.-J. Stol, W. Zhu, J. Zhu, Y. Tian, H. Liu, Automatic commit message generation: A critical review and directions for future work, IEEE Trans. Softw. Eng. (2024).

[46] T. Hoang, H.J. Kang, D. Lo, J. Lawall, Cc2vec: Distributed representations of code changes, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 518–529.

[47] S. Iyer, I. Konstas, A. Cheung, L. Zettlemoyer, Summarizing source code using a neural attention model, in: 54th Annual Meeting of the Association for Computational Linguistics 2016, Association for Computational Linguistics, 2016, pp. 2073–2083.

[48] M. Allamanis, H. Peng, C. Sutton, A convolutional attention network for extreme summarization of source code, in: International Conference on Machine Learning, PMLR, 2016, pp. 2091–2100.

[49] Y. Liang, K. Zhu, Automatic generation of text descriptive comments for code blocks, in: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 32, (no. 1) 2018.

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

[51] A. Mastropaolo, S. Scalabrino, N. Cooper, D.N. Palacio, D. Poshyvanyk, R. Oliveto, G. Bavota, Studying the usage of text-to-text transfer transformer to support code-related tasks, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering, ICSE, IEEE, 2021, pp. 336–347.

[52] S. Gao, C. Gao, Y. He, J. Zeng, L. Nie, X. Xia, M. Lyu, Code structure–guided transformer for source code summarization, ACM Trans. Softw. Eng. Methodol. 32 (1) (2023) 1–32.

[53] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, X. Liao, Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–13.

[54] J. Xu, Z. Cui, Y. Zhao, X. Zhang, S. He, P. He, L. Li, Y. Kang, Q. Lin, Y. Dang, et al., UniLog: Automatic logging via LLM and in-context learning, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–12.

[55] A. Hindle, E.T. Barr, M. Gabel, Z. Su, P. Devanbu, On the naturalness of software, Commun. ACM 59 (5) (2016) 122–131.

[56] S. Nguyen, C.T. Manh, K.T. Tran, T.M. Nguyen, T.-T. Nguyen, K.-T. Ngo, H.D. Vo, ARist: An effective API argument recommendation approach, J. Syst. Softw. (2023) 111786.

[57] Y. Dong, G. Li, Z. Jin, CODEP: grammatical seq2seq model for general-purpose code generation, in: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023, pp. 188–198.

[58] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li, et al., Codegeex: A pre-trained model for code generation with multilingual benchmarking on Humaneval-X, in: Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, 2023, pp. 5673–5684.

[59] R. Pan, A.R. Ibrahimzada, R. Krishna, D. Sankar, L.P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, R. Jabbarvand, Lost in translation: A study of bugs introduced by large language models while translating code, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.

[60] J. Zhang, P. Nie, J.J. Li, M. Gligoric, Multilingual code co-evolution using large language models, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 695–707.

[61] S. Tipirneni, M. Zhu, C.K. Reddy, Structcoder: Structure-aware transformer for code generation, ACM Trans. Knowl. Discov. Data 18 (3) (2024) 1–20.

[62] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, Q. Wang, Fill in the blank: Context-aware automated text input generation for mobile gui testing, in: 2023 IEEE/ACM 45th International Conference on Software Engineering, ICSE, IEEE, 2023, pp. 1355–1367.

[63] Y. Lan, Y. Lu, Z. Li, M. Pan, W. Yang, T. Zhang, X. Li, Deeply reinforcing android GUI testing with deep reinforcement learning, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–13.

[64] Y. Li, S. Wang, T.N. Nguyen, S. Van Nguyen, Improving bug detection via context-based code representation learning and attention-based neural networks, Proc. ACM Programm. Lang. 3 (OOPSLA) (2019) 1–30.

[65] S. Nguyen, T.-T. Nguyen, T.T. Vu, T.-D. Do, K.-T. Ngo, H.D. Vo, Code-centric learning-based just-in-time vulnerability detection, J. Syst. Softw. 214 (2024) 112014.

[66] F. Qiu, Z. Liu, X. Hu, X. Xia, G. Chen, X. Wang, Vulnerability detection via multiple-graph-based code representation, IEEE Trans. Softw. Eng. (2024).

[67] T.-T. Nguyen, H.D. Vo, Context-based statement-level vulnerability localization, Inf. Softw. Technol. 169 (2024) 107406.

[68] S. Nguyen, T.-T.Y. Vu, D.-H. Vo, VFFINDER: A graph-based approach for automated silent vulnerability-fix identification, in: Proceedings of the 15th IEEE International Conference on Knowledge and Systems Engineering, 2023.

[69] C.S. Xia, Y. Wei, L. Zhang, Automated program repair in the era of large pre-trained language models, in: 2023 IEEE/ACM 45th International Conference on Software Engineering, ICSE, IEEE, 2023, pp. 1482–1494.

[70] H. Ruan, Y. Noller, S. Tizpaz-Niari, S. Chattopadhyay, A. Roychoudhury, Timing side-channel mitigation via automated program repair, ACM Trans. Softw. Eng. Methodol. (2024).