Automated Software Engineering (2025) 32:32
https://doi.org/10.1007/s10515-025-00505-9

Tab: template-aware bug report title generation via two-phase fine-tuned models

Xiao Liu[1] · Yinkang Xu[1] · Weifeng Sun[1] · Naiqi Huang[1] · Song Sun[2] · Qiang Li[1] · Dan Yang[3] · Meng Yan[1]

Received: 10 December 2024 / Accepted: 2 March 2025
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2025

Abstract

Bug reports play a critical role in the software development lifecycle by helping developers identify and resolve defects efficiently. However, the quality of bug report titles, particularly in open-source communities, can vary significantly, which complicates the bug triage and resolution processes. Existing approaches, such as iTAPE, treat title generation as a one-sentence summarization task using sequence-to-sequence models. While these methods show promise, they face two major limitations: (1) they do not consider the distinct components of bug reports, treating the entire report as a homogeneous input, and (2) they struggle to handle the variability between template-based and non-template-based reports, often resulting in suboptimal titles. To address these limitations, we propose TAB, a hybrid framework that combines a *Document Component Analyzer* based on a pre-trained BERT model and a *Title Generation Model* based on CodeT5. TAB addresses the first limitation by segmenting bug reports into four components-*Description*, *Reproduction*, *Expected Behavior*, and *Others*-to ensure better alignment between input and output. For the second limitation, TAB uses a divergent approach: for template-based reports, titles are generated directly, while for non-template reports, DCA extracts key components to improve title relevance and clarity. We evaluate TAB on both template-based and non-template-based bug reports, demonstrating that it significantly outperforms existing methods. Specifically, TAB achieves average improvements of 170.4–389.5% in METEOR, 67.8–190.0% in ROUGE-L, and 65.7–124.5% in chrF(AF) compared to baseline approaches on template-based reports. Additionally, on non-template-based reports, TAB shows an average improvement of 64% in METEOR, 3.6% in ROUGE-L, and 14.8% in chrF(AF) over the state-of-the-art. These results confirm the robustness of TAB in generating high-quality titles across diverse bug report formats.

Keywords Bug reports · Title generation · Pre-trained model

Xiao Liu and Yinkang Xu have contributed equally to this work.

Extended author information available on the last page of the article

1 Introduction

Bug reports are essential artifacts in the software development lifecycle, playing a crucial role in maintaining software quality and usability. These reports enable users to communicate issues they encounter while using a software system, helping developers identify and resolve defects. A typical bug report contains key information, including a description of the encountered problem, the expected behavior of the software, the steps to reproduce the bug, and any additional user-provided suggestions (Bettenburg etal. 2008). Well-constructed bug reports significantly contribute to the efficiency of software maintenance and enhancement processes.

Given the central role of bug reports in software development, the quality of their titles becomes particularly important. The title of a bug report serves as its first point of entry for developers, and a concise, descriptive title can expedite the bug triage and resolution processes. Conversely, low-quality titles, and reports in general, increase the cognitive load on developers and hinder efficient bug resolution (Chaparro etal. 2017, 2019; Davies and Roper 2014; Karim etal. 2017). Recognizing this, prior research has focused on enhancing the overall quality of bug reports, particularly by improving the clarity and informativeness of bug report titles. These efforts aim to reduce the time and effort developers spend on identifying and addressing software issues, ultimately improving software development efficiency.

Bug report titles can be explicitly analyzed to streamline software engineering workflows by quickly conveying critical information to developers (Ko etal. 2006). However, in open-source communities, the quality of bug report titles can vary significantly, leading to inefficiencies in the bug triage process. To address this challenge, Chen etal. (2020) proposed an automated method called iTAPE, which focuses on generating accurate titles for bug reports on platforms like GitHub. The core idea of iTAPE is to treat bug report title generation as a one-sentence summarization task. In this approach, the content of the bug report is provided as input, and iTAPE employs a sequence-to-sequence (seq2seq) model to automatically generate a title. iTAPE leverages advanced natural language processing techniques, such as attention and encoder-decoder architectures, commonly used in machine translation tasks. This alignment of title generation with translation tasks allows iTAPE to achieve promising initial results.

Despite these advances, the performance of iTAPE remains limited when evaluated against standard metrics. Our analysis of iTAPE identifies two major performance bottlenecks that hinder further progress.

Limitation L1: Inadequate Consideration of Bug Report Components. Prior studies, such as the work by Ko etal. (2006), highlight that well-structured bug report titles generally contain three critical elements: (1) an entity or behavior of the software (*e.g.,* a user interface component or a computational function), (2) a description of the inadequacy or defect, and (3) the execution context in which the problem occurred. Furthermore, many bug tracking systems, such as Mozilla's Bugzilla, require users to adhere to structured templates when submitting bug reports, ensuring that the reports are more complete, well-organized, and easier to process. While iTAPE utilizes a seq2seq model that has proven effective in various natural

language processing tasks, its formulation of the bug report title generation task as a typical machine translation problem overlooks the nuanced structure of bug reports. By treating the entire bug report as raw input and directly generating a title, iTAPE fails to differentiate between the distinct components of the report, each of which contributes uniquely to the description of the bug. This simplification reduces the ability of the model to accurately capture and emphasize the key information needed for a meaningful and concise title, ultimately limiting its overall performance.

Limitation L2: Variability in Bug Report Styles and Template Adherence. In real-world scenarios, bug reports are written by a diverse range of users and developers, each with varying levels of expertise and different writing styles (Zhang et al. 2017). This variation in style and content can make bug reports difficult to interpret, especially when natural language is used inconsistently across reports. Although platforms like GitHub recommend using predefined templates for bug reporting, it is challenging to ensure that users adhere strictly to these guidelines. For instance, a study by Li et al. (2023) found that in 2020, only approximately 30% of bug reports submitted on GitHub followed the recommended template. Consequently, a large proportion of bug reports remain unstructured, with inconsistent or disorganized information, making them harder to process. Treating template-based and non-template-based bug reports uniformly when generating titles introduces significant challenges. The information in non-template reports is often more chaotic compared to the structured data in template-based reports, leading to inconsistencies in model training and output. Using a single model to generate titles for both types of reports is suboptimal, as the model struggles to handle the differences in structure and information quality. Furthermore, training the model on mixed datasets containing both types of reports can hinder the model's ability to effectively learn meaningful semantic patterns, ultimately resulting in unsatisfactory title generation for both template-based and non-template-based bug reports.

Addressing Limitation 1: Structured Dataset and Template-Aware Model Training. To address the **Limitation L1**, we construct a dataset consisting of 28,273 template-based issues. This dataset ensures that each bug report follows a consistent structure, making it easier for the model to identify and utilize the critical components necessary for effective title generation. To enhance the model's ability to capture the semantic relationships within bug reports, we divide the report content into four distinct categories: *Description*, *Reproduction*, *Expected behavior*, and *Others*.

We chose these four components based on the following considerations:

- **Simplification and Focus**: These four components cover the core information found in most bug reports in open-source communities, effectively helping the model understand the basic context of the issue and generate relevant and concise titles. In contrast, components like *stack trace* and *log* typically contain more technical details that are valuable for debugging, but are not directly relevant to the task of title generation.
- **Generality**: Our goal is to create a model capable of handling common bug report templates, which typically include descriptions of the issue, reproduction steps, expected behavior, and additional context. *Stack traces* and *logs*,

due to their diverse content and formatting, lack the universality needed to be included as part of the core components.

This classification enables the model to align its input with the appropriate output more effectively, allowing it to generate more meaningful and accurate titles by focusing on the specific role of each component in the bug report.

Given the strong prior knowledge and semantic understanding that pre-trained models possess, we select CodeT5 as the base model for the title generation task. CodeT5, being pre-trained on a large corpus of code and natural language, provides a robust foundation for learning the subtle associations between the different components of bug reports and the titles they require. By leveraging the model's existing capabilities in language understanding, our approach ensures that the model captures the nuances of bug report content, leading to improvements in title generation performance.

Addressing Limitation 2: Divergent Title Generation Framework. To mitigate the challenges posed by the variability in bug report styles, particularly between template-based and non-template-based reports, we introduce a divergent title generation framework. For template-compliant bug reports, our approach directly applies the bug report title generator to produce accurate and relevant titles. This ensures that reports adhering to a predefined structure are efficiently processed by leveraging the clarity and consistency of their content. For non-template-based bug reports, which tend to exhibit more variability in their structure and content, we design a document component analyzer. This analyzer effectively extracts the four key components–*Description*, *Reproduction*, *Expected behavior*, and *Others*–from unstructured reports. By identifying and classifying these components, we enable the model to align the extracted content with the title generation process. Once the key elements are extracted, the title generator utilizes this structured information to produce coherent and informative titles, improving the overall accuracy and relevance of title generation for non-template-based reports.

In conclusion, we propose an automated bug report title generation framework, named TAB. For a given bug report, if it adheres to a predefined bug report template, TAB directly leverages a fine-tuned title generator to produce the corresponding title. If the bug report does not follow the template, TAB employs a document component analyzer to assign component labels to each line of the report. The labeled content is then processed by the title generator to create a suitable title based on the identified components. To evaluate the effectiveness of TAB we conducted experiments on both template-based and non-template-based datasets. For the template-based dataset, we compared our method with baselines such as iTAPE and NNGen. Our approach achieved average improvements over the baselines of 170.4% to 389.5% in METEOR, 67.8% to 190.0% in ROUGE-L, and 65.7% to 124.5% in chrF(AF). For the non-template-based dataset, we compared our method with iTAPE, and our approach showed an average improvement of 64% in METEOR, 3.6% in ROUGE-L, and 14.8% in chrF(AF). These results demonstrate that TAB significantly outperforms existing methods in generating titles for template-based bug reports and delivers satisfactory results for non-template-based bug reports, outperforming the current state-of-the-art approaches.

This further proves the effectiveness of both our title generator and document component analyzer.

Novelty & Contributions. To sum up, the contributions of this paper are as follows:

- **Template-Based Bug Report Dataset.** We conduct a comprehensive analysis of the content and structure of GitHub issue templates specifically designed for bug reporting. Based on this analysis, we construct a dataset containing over 28,000 template-based bug reports, each adhering to a standardized format. This dataset provides a valuable resource for training and evaluating automated bug report title generation systems by ensuring consistent and structured input. To the best of our knowledge, this is the first large-scale dataset composed exclusively of template-based bug reports, offering a unique benchmark for research in this area.
- **Novel Framework.** We introduce TAB, a hybrid framework that combines the strengths of two key components: the *Document Component Analyzer* (DCA), powered by a pre-trained BERT model, and the *Title Generation Model* (TGM), built on the pre-trained CodeT5 model. The framework is specifically tailored to the unique characteristics of bug report title generation, enabling efficient and accurate title creation by leveraging both semantic understanding and contextual alignment of bug report content.
- **Extensive Evaluation.** We conduct an extensive evaluation of TAB through comprehensive experimental studies. The results demonstrate that TAB significantly outperforms two baseline approaches, consistently producing high-quality titles for semi-structured bug reports. These findings highlight TAB's ability to handle the variability in bug report structures while maintaining accuracy and relevance in title generation.
- **Open Science.** To promote transparency and reproducibility in research, we contribute to the open science community by releasing the following resources (Anonymous 2024): (1) the template-based bug report dataset used in our experiments, which contains over 28,000 structured bug reports, and (2) the implementation of the TAB framework, including both the *Document Component Analyzer* and *Title Generation Model*. These resources are made publicly available to facilitate further research in automated bug report title generation and to encourage benchmarking across different methodologies.

2 Motivation

Figures 1 and 2 illustrate the motivation behind our research. In Fig. 1, when the entire bug report is treated as a homogeneous input and a title is generated without considering the individual components, TAB produces the title "React App fails to load on IE11." While this title captures part of the context, it fails to fully convey the core issue described in the bug report. However, by leveraging the distinct components of the bug report–namely, *Description*, *Reproduction*, *Expected Behavior*, and *Others*–TAB generates the title "Unable to get property 'root' of undefined or null

<Description> **Bug Report**: the app fails to load and throws an error unable to get property root of undefined or null reference for almost all components it s working well on chrome and safari but the problem is on ie11 . <Reproduction> steps 1 create a react app and add some mui components to it 2 run the app on ie . the app should load and render components . <Expected Behavior> i m trying to come up with a solution to get rid of this error and run my app normally tech version material ui v version react v version browser ie11 . <Others>
Ground-Title:Unable to get property 'root' of undefined or null reference
Title Generated By Non-Template-based Bug-Report: React App fails to load on IE11
Title Generated By Template-based Bug Report:Unable to get property 'root' of undefined or null reference

Fig. 1 Comparison of Generated Titles for Template-Based Bug Reports

<Others> **Bug Report**: having to run ` llc ` and ` gcc ` to create a usable binary after running ` rustc ` is a lot of work . we should use llvm 's mcstreamer framework to emit object files directly . <Description>
Ground-Title:use mcstreamer to emit object files
Title Generated By Non-Template-based Bug-Report: llc and gcc should be able to create a usable binary
Title Generated By Template-based Bug Report:use llvm's mcstreamer framework to emit object filesUnable to get property 'root' of undefined or null reference

Fig. 2 Comparison of Generated Titles for Non-Template-Based Bug Reports

reference," which more accurately reflects the actual problem encountered by the user. This comparison demonstrates that accounting for the various components of a bug report significantly enhances the model's ability to capture critical information, resulting in more precise and relevant titles. This motivates the need for template-aware title generation to improve the performance of bug report summarization.

While the template-based approach improves title generation, a significant challenge remains: most bug reports, especially in open-source repositories, are unstructured and lack clearly delineated components. Manually separating these components is not only labor-intensive but also impractical at scale. As a result, existing methods have struggled to generate effective titles for non-template-based bug reports. To address this issue, we propose the use of a `Document Component Analyzer`, which automatically identifies and extracts key components from unstructured bug reports, converting them into a structured, template-like format. Figure 2 demonstrates the effectiveness of the DCA. Without DCA, TAB generates the title "llc and gcc should be able to create a usable binary," which, while relevant,

does not fully capture the core issue. However, after applying DCA to template the bug report, TAB generates the title "use llvm's mcstreamer framework to emit object files," which closely matches the original problem. This preliminary result highlights the effectiveness of the DCA in transforming non-template-based bug reports into structured ones, enabling more accurate and meaningful title generation. Thus, the DCA plays a crucial role in bridging the gap between unstructured bug reports and the need for structured, template-aware processing in automated bug report title generation.

3 Approach

3.1 Overview

To enhance the quality of bug report title generation by leveraging template-based reports, we propose a generalized approach, TAB. As shown in Fig. 3, TAB is designed with two key phases: an offline training phase and an online inference phase. The offline training phase is further divided into two critical stages: training the *Document Component Analyzer* (DCA) and constructing the *Title Generation Model* (TGM).

In the *DCA training stage*, we collect a dataset of 28,273 template-based issues from starred GitHub repositories. The DCA component uses BERT (Devlin etal. 2018) as the foundational pre-trained model, which we fine-tune specifically for this task. The fine-tuning process enables the DCA to effectively identify and label the key components of a bug report–such as *Description*, *Reproduction*, *Expected*

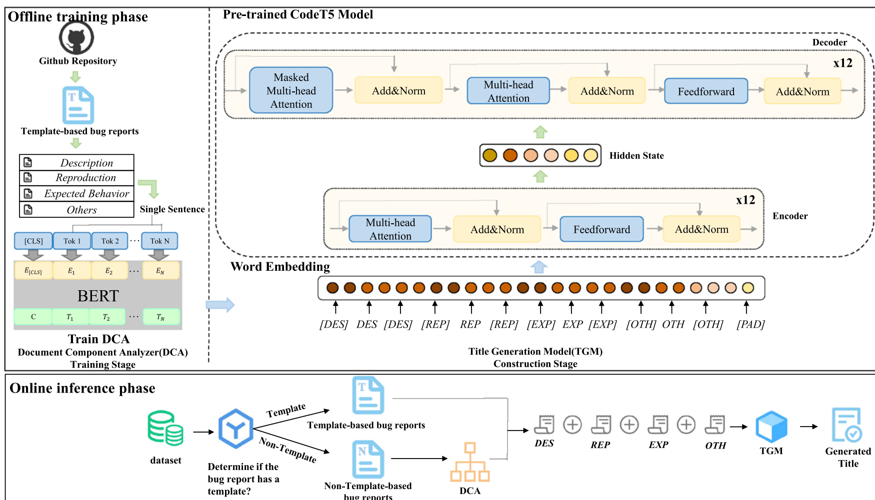

Fig. 3 The overall framework of our approach. *Offline training phase*: Train Document Component Analyzer(DCA) and Construct Title Generation Model(TGM). *Online inference phase*: Trained model is applied to generate titles for template-based bug reports

Behavior, and *Others*–transforming unstructured reports into structured, template-based formats. This transformation is crucial for improving the subsequent title generation process, especially when handling non-template-based bug reports.

In the *TGM construction stage*, we introduce four special subsegments and prompt tokens to enhance the model's understanding of the different components within a bug report. This segmentation ensures that the model can align its generated titles with the respective content more precisely. To achieve this, we fine-tune the pre-trained CodeT5 model (S. Wang etal. 2021), which is specifically designed for code-related natural language processing tasks. By leveraging CodeT5's advanced capabilities in both code understanding and natural language processing, the TGM is trained to learn effective title generation patterns, ensuring that the output titles are both relevant and concise.

The combination of the DCA and TGM allows TAB to process both template-based and non-template-based bug reports effectively. During the *online inference phase*, the system first applies the DCA to analyze and label bug report components (if necessary), followed by the TGM generating an appropriate title based on the structured input. This two-step process ensures that TAB can accurately capture the key information from diverse bug reports, improving the overall performance and relevance of the generated titles.

3.2 TAB for bug report title generation

In Fig. 3, we present the TAB framework for bug report title generation. In the remainder of this section, we will provide detailed information on the two main phases, as well as the specific customization settings for this task.

3.2.1 Dataset construction

We focus on the top 1,000 most-starred repositories, which are typically well-maintained and actively used, allowing us to gather high-quality data. From these repositories, we collected a total of 28,273 template-based issues, forming the basis of our dataset. Duplicate bug reports are a common phenomenon and could potentially affect experimental results. To address this, we compared the training set with the validation set and the training set with the test set to identify any duplicate bug reports. Our analysis revealed that the proportion of duplicate bug reports between the training set and the validation set, as well as between the training set and the test set, is less than 0.1%. This indicates that the dataset is not significantly affected by duplicate reports, ensuring no data leakage or cross-contamination between sets.

❶ *Template-Based Issue Selection*: Although GitHub introduced issue and pull request templates in March 2016, not all repositories adopt these custom templates. To identify repositories that utilize custom templates, we examine the top 1000-starred repositories for the presence of a `.github` folder, which typically contains configuration files for GitHub actions and templates. Specifically, issue templates are generally stored in the `.github/ISSUE_TEMPLATE` folder. Repositories containing this folder are flagged as using custom issue templates, allowing

us to select those repositories that have integrated this feature. As a result, we identified 503 repositories that utilize issue templates, from which we collected 28,273 template-based issues.

❷ *Data Processing*: To prepare the data for analysis, we implement a series of preprocessing steps using the Natural Language Toolkit (NLTK). First, we split the text of each bug report into sentences, ensuring clear segmentation of the content. Due to the limited input length that the model can accept, we aim to extract the most information-dense parts from the bug reports. To minimize noise, we filter out sentences containing 1) URLs, 2) `@name` mentions, and 3) markdown headlines, as these elements generally do not contribute directly to the core bug description. Although some of these sentences may contain potentially useful information, such as relevant context or error details, they are often less dense in terms of core bug content. Given the model's input length constraint, we prioritize retaining sentences that contain higher information density. Consequently, we remove these specific sentences to ensure the model focuses on the most relevant aspects of the bug report. URLs and user mentions are excluded because our focus is on summarizing the bug report's main content, while markdown headlines are removed to avoid introducing a high number of out-of-vocabulary (OOV) words. Next, we tokenize the text using NLTK, which has been shown in previous research to outperform other common NLP libraries in tokenizing software documentation. In addition, version numbers (*e.g.,* "1.2.3") are normalized to "version" and numeric values are replaced with "0" to reduce variability in the dataset. Any tokens containing non-ASCII characters are removed to ensure consistency, and texts with more than 50% non-ASCII tokens are flagged as "nonASCII". This preprocessing ensures a clean and uniform dataset for further analysis. For the template-based issues, we further extract key fields such as *Description*, *Reproduction*, *Expected Behavior*, and *Others* to construct a well-structured corpus. This division of content into distinct fields allows us to leverage the unique components of each issue for more precise and meaningful title generation.

3.2.2 Offline training phase

In this phase, we describe the process of training the Document Component Analyzer (DCA) and constructing the Title Generation Model (TGM). The DCA is designed to categorize the different components of bug reports, while the TGM is responsible for generating relevant and concise titles based on these structured components.

❶ *DCA Training*: The Document Component Analyzer (DCA) is built on BERT (Devlin etal. 2018), a pre-trained Transformer model known for its deep semantic understanding capabilities. DCA fine-tunes BERT for the specific task of categorizing bug report sentences into key components, ensuring the model can effectively differentiate between the various aspects of a bug report.

For this task, we first preprocess the bug reports by segmenting them into sentences. Each sentence is categorized into one of four classes: *Description* (`des`), *Reproduction* (`rep`), *Expected Behavior* (`exp`), and *Others* (`oth`). This categorization is essential for accurately structuring the bug report, as each component serves a different function in describing the issue and providing context. Given an input

sentence $S = [w_1, w_2, \dots, w_m]$, we first tokenize it using the BERT tokenizer, which transforms the sentence into a sequence of tokens:

$$T = [\texttt{[CLS]}, t_1, t_2, \dots, t_n, \texttt{[SEP]}] \quad (1)$$

where `[CLS]` is a special token representing the entire sentence, `[SEP]` is a separator token marking the end of the sentence, and t_i represents each token in the sentence.

Once tokenized, these tokens are converted into embeddings, where each token t_i is mapped to an embedding vector x_i. This embedding sequence is then passed through the multiple layers of the BERT model, where self-attention mechanisms capture the contextual information of each token within the sentence. The final hidden state of the `[CLS]` token, $h_{\texttt{[CLS]}}$, is extracted, as it encodes the semantic representation of the entire sentence. This representation is fed into a fully connected layer followed by a softmax function to predict the class of the sentence:

$$P(y|h_{\texttt{[CLS]}}) = \mathrm{softmax}(W \cdot h_{\texttt{[CLS]}} + b) \quad (2)$$

where W and b are learnable parameters, and y represents the predicted class (*i.e.,* one of the four bug report components). This fine-tuned DCA can accurately classify sentences into the correct bug report components, allowing for better-structured inputs for the Title Generation Model (TGM).

❷ *TGM Construction*: At this stage, our primary goal is to train the Title Generation Model (TGM), which generates concise and relevant titles based on the distinct textual components of bug reports, including the *description*, *reproduction*, *expected behavior*, and *others*. These components are distinguished and integrated using prompt-based learning. Our model is built on top of the pre-trained (S. Wang etal. 2021) architecture, which has been fine-tuned using our custom dataset. It is important to note that while we utilize CodeT5, TAB can also be adapted to other Pre-trained Code Models (PCMs), as discussed in Sect. 5.3.

CodeT5 follows the encoder-decoder architecture of T5, employing denoising sequence-to-sequence tasks during pre-training. Additionally, it incorporates two identifier-related tasks–identifier tagging and masked identifier prediction–which allow it to integrate the semantics of developer-assigned identifiers. These tasks enable CodeT5 to effectively capture the contextual significance of identifiers and better understand dependencies between them. CodeT5 has demonstrated high adaptability across various downstream tasks, making it an ideal choice for title generation in software engineering contexts.

(1) *Prompt Design*: Liu P etal. (2023) modifies the original input by introducing specific prompt templates designed to guide the model in generating more accurate outputs. However, creating effective prompts for downstream tasks can be challenging and often requires careful design and experimentation. In our approach, we design four distinct input slots to represent the key components of a bug report: the *description*, *reproduction*, *expected behavior*, and *others*. These slots are denoted by placeholders in the prompt. The structure of the prompt for the title generation task is defined as follows:

$$f_{\text{Input}} = \texttt{DES} : [X] : \texttt{REP} : [Y] : \texttt{EXP} : [Z] : \texttt{OTH} : [V] \tag{3}$$

Here, $[X]$, $[Y]$, $[Z]$, and $[V]$ are placeholders for the respective bug report components. This prompt structure ensures that the model is provided with clear and structured information about each aspect of the bug report, facilitating more accurate title generation.

(2) *Prompt Tuning on CodeT5*: CodeT5's encoder consists of multiple layers of Transformers, each comprising a self-attention layer and a feed-forward network. The encoder's role is to generate contextual embeddings for the input sequences, which in our case are the bug report components and prompt tokens. We initialize the encoder with a pre-trained CodeT5 model to take advantage of its strong contextual understanding, specifically in the domain of software-related natural language tasks.

- ***Encoder***: During training, the model takes pairs of sub-segments (representing the bug report components) and prompt tokens as input, denoted by *TG*. These inputs are tokenized using a subword (Sennrich etal. 2015), which helps mitigate out-of-vocabulary (OOV) issues by breaking complex identifiers into subtokens. By retaining the original tokenization vocabulary from the pre-trained CodeT5 model, we ensure that the model inherits its semantic knowledge and starts from a strong initialization point, allowing it to effectively learn title generation patterns. The tokenized input sequence is then passed through the embedding layer, where each token is mapped to an embedding vector $\tilde{\mathbf{X}} = \{\tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_2, \ldots, \tilde{\mathbf{x}}_n\}$. These embeddings are processed through the stacked layers of the CodeT5 encoder, where each Transformer block contains a multi-headed self-attention mechanism (J. Devlin etal. 2019), a feed-forward network, and layer normalization (Ba etal. 2016). The process is as follows:

$$\hat{\mathbf{X}} = \text{MultiHead}(\tilde{\mathbf{X}}) \tag{4}$$

$$\mathbf{X}^i = \text{LayerNorm}(\hat{\mathbf{X}} + \text{FFN}(\hat{\mathbf{X}})) \tag{5}$$

In the above equations, MultiHead(·), FFN(·), and LayerNorm(·) represent the multi-head self-attention layer, the feed-forward network, and the layer normalization operation, respectively. The index i denotes the output of the i-th Transformer layer. The multi-head self-attention mechanism captures long-range dependencies between the tokens, allowing the model to understand complex relationships within the input. The feed-forward network enhances feature extraction by linearly transforming the token embeddings, and layer normalization ensures stable training by normalizing the token embedding distributions.

After processing through l Transformer layers, the input sequence *TG* is encoded into a sequence of contextual embeddings $\mathbf{X}^l = \{\mathbf{x}_1^l, \mathbf{x}_2^l, \ldots, \mathbf{x}_n^l\}$. The last hidden state x_n^l is used as the final contextual vector representa-

tion $\mathbb{R}$ of the input *TG*. This representation serves as the foundation for generating accurate and semantically meaningful titles for the bug reports.

- *Decoder*: The process of generating a bug report title follows a similar approach to that of sequence generation tasks. Specifically, the Title Generation Model (TGM) is designed to generate the corresponding title one token at a time, conditioned on the input components of the bug report (*i.e.,*, description, reproduction steps, expected behavior, and others). The generation of the new title t' is performed sequentially, where each token t'_i is generated based on the previously generated tokens and the input components. Formally, the task of generating a bug report title can be defined as finding the sequence $\vec{t'}$ such that:

$$\vec{t'} = \arg\max_{\vec{t'}} P_\theta(t' | I_u)$$

where $P_\theta(t' | I_u)$ represents the probability of the title sequence t' given the input components I_u. This probability can be factorized as:

$$P_\theta(t' | I) = \prod_{i=1}^{w} P_\theta(t'_i | t'_1, ..., t'_{i-1}, x, x', e, t)$$

where $P_\theta(t'_i | t'_1, ..., t'_{i-1}, x, x', e, t)$ represents the probability of generating token t'_i conditioned on the previously generated tokens and the input components of the bug report. The model is trained to minimize the negative log-likelihood of the predicted title sequence relative to the ground-truth title. This ensures that the generated titles closely match the actual titles used in the bug reports.

The architecture of the Title Generation Model consists of two key components: the self-attention layer and the encoder-decoder attention layer. The self-attention mechanism captures dependencies between the already generated tokens, while the encoder-decoder attention mechanism aligns the input components (description, reproduction, etc.) with the tokens being generated. When calculating the attention distribution between the generated tokens y_j and the input tokens $w_1, ..., w_m$, the attention score is calculated using the key vector K from the encoder outputs $z = (z_1, ..., z_{|I|})$. The attention distribution α^j between the generated tokens y_j and the input components is computed as follows:

$$\alpha^j = \text{softmax}\left(\frac{Q_j K^T}{\sqrt{d_k}}\right)$$

where Q_j represents the query vector for the generated token y_j, and d_k is the dimensionality of the key vectors. This mechanism enables the model to effectively capture the relationships between the bug report content and the generated title, leading to coherent and contextually accurate title generation.

3.2.3 Online inference phase

In the online inference phase, the process begins by determining whether the bug report is template-based. For template-based reports, the model directly extracts the relevant components from the issue body, including the *description*, *reproduction*, *expected behavior*, and *others* segments. These components are clearly delineated in template-based reports, allowing for straightforward extraction. 1) For non-template-based reports, the process requires an additional step. We employ the Document Component Analyzer (DCA) to automatically segment the bug report into a structured format that mimics a template. The DCA identifies and categorizes the key segments, including the *description*, *reproduction*, *expected behavior*, and *others* components, ensuring that even unstructured reports can be processed consistently. 2) Once the four key segments are extracted–whether from a template-based or DCA-processed bug report–they are combined with predefined prompt tokens. These prompt tokens provide contextual cues that guide the model in understanding the relationship between the components of the report and the title it needs to generate. 3) The final step involves feeding the four extracted segments, along with the prompt tokens, into our fine-tuned CodeT5 model. The model, which has been specifically trained for bug report title generation, processes this input and generates a concise, contextually appropriate title that accurately reflects the content of the bug report. This end-to-end process ensures that the system can generate high-quality titles regardless of whether the input bug report follows a predefined template.

4 Experimental setup

This section presents our research questions, baselines, evaluation metrics and evaluation methods.

4.1 Research questions

We want to investigate the following research questions:

- RQ1: How effective is TAB in generating titles for template-based bug reports?
- RQ2: Can TAB generate appropriate titles for all bug reports?
- RQ3: How do different pre-trained models affect TAB?
- RQ4: Can TAB generate higher-quality title than state-of-the-art baselines by human study?

4.2 Baselines

To evaluate the performance of TAB, we use two baselines belonging to different types: iTAPE (Chen etal. 2020), Liu etal. (2018).

iTAPE. iTAPE is the first automated method to generate titles for bug reports, which is also a Seq2Seq summarization method. Since iTAPE formulates title generation into a one-sentence summarization task, it cannot directly use the issue body which we split as input. To address this issue, we take the issue body before the content split as the input to the iTAPE. In particular, we directly used the replication package published in the Chen etal. (2020), to ensure that the model parameters and other settings were consistent with the original.

NNGen. NNgen is a state-of-the-art commit messages generation method. It generates commit messages by using the nearest neighbor algorithm to retrieve from historical commits. In this task, we use NNgen to generate bug report titles based on the corresponding issue bodies.

4.3 Evaluation settings

Following our approach, we utilize the pretrained BERT-base-uncased model for the Document Component Analyzer (DCA) Training Stage with a learning rate of 2e-5, 5 epochs, and a batch size of 8. For the Title Generation Model (TGM) Construction Stage, we employ the pretrained CodeT5-base model with a learning rate of 5e-5, 3 epochs, and a batch size of 32. All experiments are conducted in a high-performance environment using three NVIDIA A800-SXM4-80GB GPUs.

4.4 Evaluation metrics

We evaluate the effectiveness of our approach and baselines with three metrics, ROUGE-L (Lin 2004), Lavie and Agarwal (2007) and Popović (2015). Such evaluation metrics are widely used for text generation tasks and verified to be reliable proxies (Roy etal. 2021). We obtain these metric scores using *nlg-eval*[1] (Sharma etal. 2017), *rouge*,[2] and *chrF*[3] package.

ROUGE-L. ROUGE is a set of metrics that was first introduced for summarization. Unlike BLEU which only calculates precisions, ROUGE is the harmonic mean between n-gram precisions and recalls of a generated message to the reference message. We select the ROUGE-L (*i.e.,* n-gram in ROUGE-L is the longest common subsequence) as our evaluation metrics which are also used by Liu etal. (2019).

METEOR. METEOR is proposed as a metric that correlates better at the sentence level with human evaluation. The calculation of METEOR needs to create an alignment between the generated and the reference message by mapping each unigram in the generated message to 0 or 1 unigram in the reference message. Based on this alignment, unigram precision and recall are computed. The METEOR score is the harmonic mean between precision and recall with the weight for recall 9 times

1 https://github.com/Maluuba/nlg-eval.
2 https://pypi.org/project/rouge/
3 https://github.com/m-popovic/chrF.

as high as the weight for precision. METEOR further employs a penalty factor for fragmentary matches.

chrF. chrF is an automatic evaluation metric that works solely on character n-grams rather than word n-grams. It can be seen as a character n-gram F-score.

4.5 Evaluation methods

4.5.1 Evaluation on template-based bug reports

The RQ1 aims to investigate the effectiveness of TAB in generating titles for template-based bug reports. Hence, we evaluate it and the baselines on our datasets in terms of ROUGE-L, METEOR, and chrF. The issue has the attribute of timestamp. For each GitHub project, we sorted its issues in the ascending order of the issue's creation time. We then divided the dataset by time, using the first 80% of the issues for training, while the remaining 20% were shuffled and split equally for validation and testing.

This approach was chosen to better simulate the model's application in real-world scenarios. By using earlier bug reports for training and later reports for validation and testing, we ensure the model can handle real-time issues and generalize well to new, unseen problems. This method also helps avoid data leakage and prevents overlap between the training and test sets, thereby improving the reliability of the experimental results. Although this strategy may introduce shifts in bug report characteristics over time, it allows us to assess the model's performance when dealing with new and evolving issues.

Since our datasets are made up of issues from multiple GitHub projects, it is necessary to evaluate the impact of cross-project data on the quality of the title generated by TAB. Specifically, we use two validation patterns in RQ1, i.e., In-project validation and cross-project validation.

Within-project validation refers to training and testing with data from the same GitHub project. Github projects are divided into sizes, but too little issue data will make the model overfitting. Therefore, we only conduct experiments on projects with more than 5000 semi-structured issues (using the bug report template). There are 28 eligible projects. The average of the testing results of all projects are used as the in-project validation result.

Cross-project validation means the training data, validation data and test data are from all GitHub projects. The overall training set is obtained by combining the training sets of all projects. Validation set and test set are the same way.

4.5.2 Evaluation on non-template-based bug reports

The dataset introduced in iTAPE (Chen etal. 2020) contains 333,563 bug report samples from the top 200 starred repositories on GitHub, significantly larger than the dataset we use for training. A notable observation is that the majority of GitHub bug reports are non-template-based, highlighting the prevalence of unstructured

reports in real-world repositories. In RQ2, we aim to assess whether our model, TAB, is capable of generating accurate titles for non-template-based bug reports.

For template-based bug reports, TAB can directly generate titles by leveraging the clearly defined sections such as *description*, *reproduction*, and *expected behavior*. However, non-template-based bug reports typically consist of one or more paragraphs without explicit labels for these components. To address this issue, we train a `Document Component Analyzer (DCA)` using our own curated dataset of template-based bug reports. This DCA is then applied to the non-template-based bug reports, automatically classifying and extracting the key components required for title generation.

In this evaluation, we use the iTAPE dataset (Chen etal. 2020) as the test set for verifying our approach. Since the iTAPE dataset contains both template-based and non-template-based bug reports, we first divide the dataset into two subsets: template-based and non-template-based reports. For the non-template-based subset, we apply our trained DCA to extract the key components (*e.g., description*, *reproduction*, *expected behavior*) from the unstructured text, treating these extracted components as subtypes within the non-template-based bug reports.

It is important to emphasize that while iTAPE uses its own dataset for training, our DCA is trained on a separate dataset that we collected, consisting of 28,273 template-based bug reports from GitHub repositories. We do not use the iTAPE dataset for DCA training, ensuring that our DCA is generalized and capable of handling unseen non-template-based reports. After applying the DCA to the non-template-based bug reports, we evaluate the performance of TAB in generating titles and compare it against the performance of iTAPE on the same dataset. This comparison allows us to verify the robustness of TAB in handling non-template-based bug reports.

5 Results

5.1 RQ1:TAB + template-based bug reports

We evaluate the performance of TAB using two validation patterns and compare its effectiveness against baseline approaches. The results of these experiments are

Table 1 Effectiveness of TAB and baselines by using validation two patterns

Pattern	Method	METEOR		ROUGE-L		chrF(AF)	
Within	iTAPE	9.47	+181.4%	17.76	+87.6%	19.08	+77.7%
	NNGen	6.32	+321.7%	13.66	+143.9%	14.28	+137.5%
	TAB	**26.65**	–	**33.31**	–	**33.91**	–
Cross	iTAPE	10.03	+170.4%	19.58	+67.8%	20.26	+65.7%
	NNGen	5.54	+389.5%	11.33	+190.0%	14.95	+124.5%
	TAB	**27.12**	–	**32.86**	–	**33.57**	–

Bold value indicates the highest score in the same dataset and for the same evaluation metric

summarized in Table 1. The “Within” column represents the results obtained using the within-project validation pattern, where both training and testing data are from the same project. This approach assesses how well TAB can generalize within a single project. The “Cross” column shows the results of the cross-project validation pattern, where the model is trained on one set of projects and tested on entirely different projects. This validation pattern evaluates TAB’s ability to generalize across diverse projects, which is critical for demonstrating the robustness and scalability of the model in real-world, multi-project environments. By comparing the results across these two validation patterns, we can assess the strengths and weaknesses of TAB relative to the baseline models in both intra- and inter-project scenarios, providing a comprehensive understanding of its performance in practical settings.

Table 1 presents the performance comparison of TAB against two baseline methods, iTAPE and NNGen, under two validation patterns: within-project and cross-project. The results are evaluated using three metrics: METEOR, ROUGE-L, and chrF.

Within-Project Validation. In the within-project validation, where training and testing are performed on the same project, TAB significantly outperforms the baselines across all metrics. For instance, TAB achieves a METEOR score of 26.65, compared to 9.47 for iTAPE and 6.32 for NNGen. This represents a substantial improvement, demonstrating that TAB is highly effective at generating accurate titles when both the training and testing data are from the same project. Similarly, TAB achieves the highest ROUGE-L score of 33.31, which is considerably higher than the 17.76 for iTAPE and 13.66 for NNGen. The chrF(AutoEval Framework) scores also indicate a clear advantage for TAB, with a score of 33.91, compared to 19.08 and 14.28 for iTAPE and NNGen, respectively. These results show that TAB not only captures the overall content more effectively but also generates titles that are closer to the ground-truth in terms of fluency and accuracy.

Cross-Project Validation. In the more challenging cross-project validation, where training is done on one set of projects and testing on a completely different set, TAB continues to demonstrate strong performance. It achieves a METEOR score of 27.12, outperforming both iTAPE (10.03) and NNGen (5.54). This indicates that TAB generalizes well across different projects, a crucial capability for real-world applications where bug reports often come from a variety of projects. For the ROUGE-L metric, TAB achieves a score of 32.86, surpassing iTAPE’s 19.58 and NNGen’s 11.33. This suggests that TAB maintains its ability to generate titles that closely reflect the key content of bug reports, even in cross-project scenarios. The chrF(AutoEval Framework) score further emphasizes this point, with TAB scoring 33.57, compared to 20.26 for iTAPE and 14.95 for NNGen. These results demonstrate that TAB is capable of handling the variability between projects, maintaining high accuracy and fluency in title generation.

Overall, the experimental results clearly show that TAB outperforms both iTAPE and NNGen across all evaluation metrics and in both validation patterns. While iTAPE shows a reasonable performance improvement over NNGen, it still falls significantly behind TAB in all aspects. Notably, NNGen struggles the most, particularly in cross-project validation, where its scores are considerably lower than both iTAPE and TAB. The significant gains made by TAB in both METEOR and

ROUGE-L suggest that it is particularly adept at capturing the semantic meaning and key content of bug reports, while its high chrF scores indicate that the generated titles are more fluently aligned with the ground-truth. These results highlight the robustness and versatility of TAB in generating high-quality bug report titles, regardless of whether the reports are from the same or different projects.

Summary: The results confirm that TAB outperforms both iTAPE and NNGen across all metrics in both within-project and cross-project validations. Notably, TAB demonstrates strong generalization capabilities, making it effective in handling diverse bug reports across different projects. This highlights the robustness and scalability of TAB for automated bug report title generation.

5.2 RQ2: TAB + non-template-based bug reports

The dataset introduced in Chen etal. (2020) is divided into two parts for our experiments. In RQ1, we focus on the analysis of template-based bug reports. For the non-template-based bug reports, we employ the `Document Component Analyzer (DCA)` as an intermediate processing layer. The `DCA` segments and classifies the unstructured bug reports into meaningful components, which are then used as input for the TAB framework. This approach allows us to directly compare the performance of TAB against iTAPE on non-template-based bug reports, highlighting the effectiveness of TAB in handling unstructured data.

Table 2 presents the comparative results between TAB and iTAPE on non-template-based bug reports. Although the performance of TAB on non-template-based bug reports is slightly lower than its performance on template-based reports, our approach still demonstrates substantial effectiveness due to its advanced architecture. Specifically, TAB achieves approximately 64% higher METEOR, 3.6% higher ROUGE-L, and 14.8% higher chrF scores compared to iTAPE. These improvements highlight the robustness and adaptability of TAB, enabling it to outperform iTAPE even when handling the more challenging and unstructured non-template-based bug reports. The results emphasize that TAB remains effective in generating accurate and fluent titles, even in less structured scenarios.

Summary: The results demonstrate that TABconsistently outperforms iTAPE on non-template-based bug reports, achieving significant improvements across all metrics. Despite the inherent challenges of unstructured data, TABshows strong adaptability and effectiveness, making it a robust solution for generating accurate titles even in less structured scenarios.

5.3 RQ3: impact of pre-trained models on TAB performance

To validate the importance of the pre-trained model CodeT5 in TAB, we conducted additional experiments by comparing it with two other prominent pre-trained models, **CodeGen** (Nijkamp etal. 2022) and **Unixcoder** (D. Guo etal. 2022), as well as the natural language model **T5** (Raffel etal. 2023). This comparison allows us to better understand how different pre-trained models affect TAB. Such an analysis

Table 2 Effectiveness of iTAPE and TAB for non-template-based bug reports

Method	METEOR	ROUGE-L	chrF(AF)
iTAPE	14.25	28.30	28.17
TAB	**23.37**	**29.31**	**32.34**

Bold value indicates the highest score in the same dataset and for the same evaluation metric

provides a comprehensive evaluation of the effectiveness of pre-trained models across various code generation tasks and offers guidance for model selection. In terms of data usage, we use template-based data(`Within-project validation and Cross-project validation`), which refers to RQ1, and non-template-based data, which refers to RQ2.

The evaluation results are shown in Table 3. As observed in Table 3, the pre-trained CodeT5 model consistently achieves the best performance in all cases. For template-based data, the use of the pre-trained CodeT5 model leads to an enhancement of the METEOR score by 3.1% to 25.8%, an increase in the ROUGE-L score by 2.8% to 164.8%, and an improvement in the chrF score by 4.3% to 17.0%. This improvement can be attributed to the pre-training process of the CodeT5 model, particularly for understanding tasks. Compared to Nijkamp etal. (2022) and D. Guo etal. (2022), CodeT5 utilizes a bidirectional Transformer structure that excels at processing complex contexts and language patterns. Additionally, CodeT5 is optimized during pre-training to capture the semantic features of code, enabling it to better grasp the logical structure and finer details. In contrast, CodeGen focuses more on generation tasks, and Unixcoder emphasizes unidirectional encoding. Compared to T5 (Raffel etal. 2023), CodeT5 employs a specialized architecture fine-tuned specifically for code-related tasks, utilizing a bi-directional converter structure that excels at capturing the intricate syntactic and semantic features of programming languages. In addition, CodeT5 is pre-trained on a large code corpus, enabling it to better understand technical terms, variable names, error messages or code snippets in bug reports. In contrast, T5 is primarily optimized for general natural language processing tasks and lacks the specific pre-training needed to capture the technical nuances and logical structure of code-related content. In contrast, CodeT5 is better at generating contextually accurate and technically precise headings and summaries, making it better suited for tasks involving both natural language and code. As a result, CodeT5 is more effective at retaining and applying structured information when generating titles, leading to significant improvements in performance metrics.

For non-template-based data, the use of the pre-trained CodeT5 model leads to an enhancement of the METEOR score by 0.2% to 20.0%, an increase in the ROUGE-L score by 1.4% to 158.2%, and an improvement in the chrF score by 0.5% to 14.8%. This is because non-template-based bug reports tend to be less structured, with more flexible and diverse content. CodeT5's pre-training process, particularly through multi-task learning, enhances its ability to handle diversity and adaptability in tasks, making it more capable of recognizing and leveraging contextual relationships in unstructured text. In contrast, CodeGen, which focuses primarily on generation tasks, struggles to fully utilize context in handling free-form text, while Unixcoder's unidirectional nature limits its ability to capture complex, non-linear relationships. Similarly, while T5 excels

at general natural language tasks, it is not optimized for handling the technical intricacies of code or domain-specific language. CodeT5, with its specialized pre-training on code data, captures the semantic nuances of code and technical terminology more effectively, allowing it to generate more contextually accurate titles for tasks that involve both natural language and code. This makes CodeT5 superior for tasks like title generation for bug reports, where understanding the technical context is critical.

Summary: Overall, the experimental results confirm that the use of the pre-trained CodeT5 model significantly enhances the model's ability to generate high-quality titles for both template-based and non-template-based bug reports, outperforming both code language models and general natural language models. CodeT5's specialized pre-training, which incorporates a deep understanding of both technical code and natural language, allows it to generate more accurate and contextually relevant titles than models focused solely on code generation or natural language processing.

5.4 RQ4: comparison with baselines via human evaluation

In this RQ, we conducted a human study to evaluate the quality of the question titles generated by the title generation baseline (i.e., iTAPE) and our proposed approach TAB. The evaluation was performed using three criteria: similarity, naturalness, and informativeness. The scoring scale of these criteria ranges from 1 to 4, where a higher score indicates a better quality of the generated titles. This human evaluation methodology has been commonly used in previous studies for similar tasks (Wei 2019), ensuring the reliability and validity of the evaluation process.

- **Similarity**. This criterion measured the degree of similarity between the generated titles and the ground truth titles. Evaluators were asked to assess how well the generated titles captured the essence of the ground-truth title and aligned with the intended meaning.

Table 3 Impact of Pre-trained Models on TAB Performance Results

Data	Pre-trained models	METEOR	ROUGE-L	chrF(AF)
Within	CodeGen	21.19	12.58	32.16
	Unixcoder	25.85	32.31	28.99
	T5	26.59	32.33	33.74
	CodeT5	**26.65**	**33.31**	**33.91**
Cross	CodeGen	21.93	12.62	32.2
	Unixcoder	26.48	31.98	28.73
	T5	26.5	32.4	33.4
	CodeT5	**27.12**	**32.86**	**33.57**
Non-template-based	CodeGen	19.78	11.35	30.85
	Unixcoder	22.46	28.56	28.16
	T5	22.39	28.06	30.96
	CodeT5	**23.37**	**29.31**	**32.34**

Bold value indicates the highest score in the same dataset and for the same evaluation metric

- **Naturalness**. The naturalness criterion focused on the grammaticality and fluency of the generated titles. Evaluators evaluated how well the titles were composed in terms of language usage, syntax, and overall coherence.
- **Informativeness**. The informativeness criterion gauged the amount of content conveyed by the generated titles. Evaluators assessed the extent to which the titles provided relevant and useful information about the question post or issue, regardless of their grammatical correctness or fluency.

We randomly selected 160 issues from the test set. For each issue, we gathered the actual title as well as two titles generated by iTAPE and TAB. To evaluate these generated titles, we enlisted four graduate students who are knowledgeable about GitHub issues but are not co-authors of the paper. Each student reviewed 40 issues using three criteria: similarity, naturalness, and informativeness. They were permitted to use the internet to investigate any unfamiliar concepts related to the issues. Furthermore, to maintain the quality of the evaluations, each student was restricted to assessing only 20 issues in a half-day session.

The evaluation results for the task are presented in Fig 4, where the performance of TAB or iTAPE is assessed based on the three evaluation criteria: similarity, naturalness, and informativeness.

In terms of the similarity criterion, TAB surpasses iTAPE with an average score of 3.6. This suggests that the titles produced by TAB are regarded as high quality and closely align with the ground-truth titles. The incorporation of title hints by developers plays a significant role in the elevated similarity scores attained by TAB.

Regarding the naturalness criterion, both TAB and iTAPE show comparable performance. This is consistent with the expectation that titles produced by deep learning models are typically readable and understandable to users.

In the context of the informativeness criterion, TAB outperforms iTAPE by producing more comprehensive titles. This indicates that TAB, by taking into account developer intent and leveraging a pre-trained CodeT5 model, is better at managing long-term dependencies in the question or issue body and demonstrates superior semantic understanding.

Given the subjectivity inherent in human evaluation, we use Fleiss Kappa (Fleiss 1971) to assess the consistency of the scoring results among the students. The overall Kappa value obtained for the task is 0.774, indicating substantial agreement among the students in their assessments. Following the scoring process, the students engaged in discussions to address their disagreements and arrive at a consensus, which helps to mitigate bias in the human evaluation of our study.

Summary: In terms of similarity, naturalness, and informativeness criteria, the human evaluation result supports the superior performance of TAB when compared to iTAPE.

6 Discussion

6.1 The impact of varying `document component analyzer(DCA)`

Given the maturity of sentence classification techniques, we chose two well-established methods–CNN (Lawrence et al. 1997) and Devlin etal. (2018)–to perform the task of sentence classification. CNN is a relatively simple yet effective method, while BERT represents a more advanced state-of-the-art approach. For the CNN model, we adopted the architecture proposed by Huang et al. (2018), which has been used to classify bug report sentences from GitHub. For BERT, we utilized BERTOverflow (Tabassum etal. 2020), a BERT model pre-trained on StackOverflow data, which has demonstrated excellent performance in software engineering tasks. We then fine-tuned BERTOverflow on sentences from template-based bug reports.

We conducted experiments on template-based bug reports to compare the performance of CNN and BERT in classifying the four types of sentences: *Description*, *Reproduction*, *Expected Behavior*, and *Others*. Table 4 presents the precision, recall, and F1-scores for both models. The results show that BERT consistently outperforms CNN across all categories, achieving higher precision, recall, and F1-scores, demonstrating its superior capability in sentence classification tasks within the software domain.

Furthermore, we investigated the impact of different classifiers on the quality of titles generated by TAB. Specifically, we used CNN and BERT to classify non-template-based bug reports, followed by the application of TAB to generate titles based on the classified components. As shown in Table 5, the quality of the classifier has a

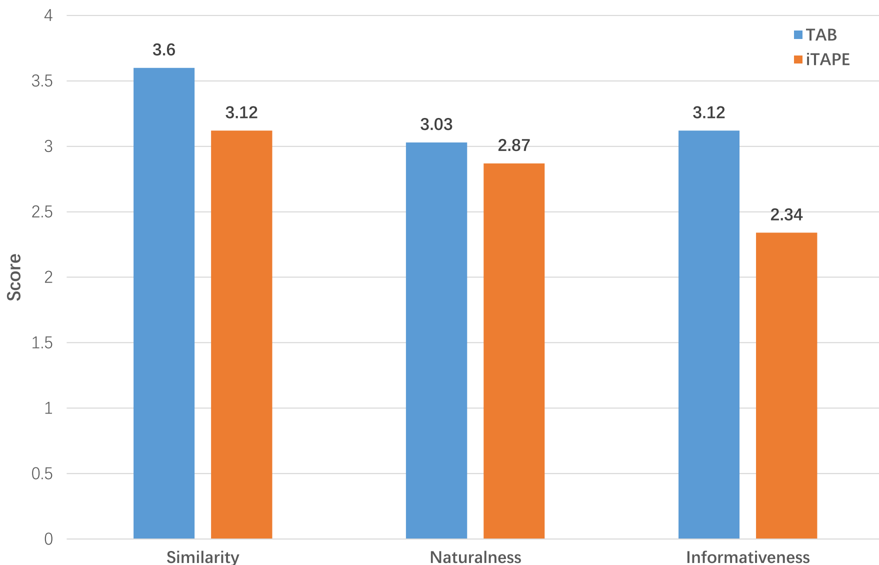

Fig. 4 The average score value of our human study by considering similarity, naturalness, and informativeness for the task

Table 4 The performance comparison of CNN and BERT

Label	Method	Precision	Recall	F1-score
Description	CNN	73.49	84.02	78.40
	BERT	**88.20**	**92.54**	**90.32**
Reproduction	CNN	85.95	88.34	87.13
	BERT	**93.01**	**93.18**	**93.09**
Expected	CNN	86.34	77.49	81.68
	BERT	**92.67**	**91.99**	**92.33**
Others	CNN	95.63	89.42	92.42
	BERT	**96.96**	**92.72**	**94.79**
Accuracy	CNN	–	–	84.91
	BERT	–	–	**92.61**

Bold value indicates the highest score in the same dataset and for the same evaluation metric

Table 5 The performance of TAB for non-templated-based bug reports with different classifiers

Classifiers	METEOR	ROUGE-L	chrF(AF)
CNN	14.97	28.73	29.09
BERT	**15.62**	**30.95**	**30.30**

Bold value indicates the highest score in the same dataset and for the same evaluation metric

direct effect on the performance of TAB. A more accurate classifier, such as BERT, improves the overall title generation quality, underscoring the importance of robust sentence classification for enhancing TAB's effectiveness.

6.2 Why our TAB works better

Our TAB demonstrates exceptional performance due to the specially designed prompt tokens and advanced title generation model, which are specifically crafted to address the complexities of bug report data. The key factors contributing to its effectiveness are:

Pre-Trained CodeT5 Model: TAB employs prompt tokens to handle different sections of bug reports, such as *descriptions*, *reproduction*, *expected behavior*, and *others*. By using prompt tokens, the system can segment the various components of the bug report, helping the model to accurately grasp the semantics and logical relationships of each part. TAB leverages the CodeT5 model, allowing it to capture deep contextual information between different sections of the bug report. This comprehensive context awareness provides CodeT5 with a significant advantage in generating high-quality titles, improving both the accuracy of the titles and ensuring that they cover the critical aspects of the bug report. By combining prompt tokens and the CodeT5 model, TAB can effectively handle complex bug reports, delivering clear and precise title generation results, thus enhancing overall issue tracking and management efficiency.

Document Component Analyzer (DCA): One of the strengths of TAB is its ability to process both template-based and non-template-based bug reports. For non-template-based data, a trained DCA is used to segment and categorize the content before it is processed by TAB. This preprocessing step helps in organizing the input data effectively, ensuring that even unstructured bug reports are formatted in a way that can be accurately processed by the `Title Generation Model(TGM)`. This structure extraction improves the overall quality of the generated titles and ensures that key bug report information is appropriately represented.

Template-Based Bug Report Generation: TAB is designed to convert non-template-based bug reports into a standardized format. This templating ensures uniformity across bug reports, making it easier for the `Title Generation Model(TGM)` to focus on extracting relevant information without being hindered by inconsistencies in bug report formatting. This transformation improves the model's ability to generalize across diverse data and boosts the performance of the title generation task, as seen in its superior handling of both structured and unstructured reports.

Scalability and Adaptability: TAB is not limited to a specific bug-tracking system or domain. Its modular design allows it to be adapted for various software engineering title generation tasks, making it highly versatile. The offline training phase, which leverages fine-tuned models like Devlin etal. (2018) and S. Wang etal. (2021), ensures that TAB can be fine-tuned for other types of tasks with minimal adjustments. This flexibility ensures that TAB is not only effective for the current dataset but can also be extended to handle other software engineering issues.

Comprehensive Coverage of Bug Reports: By splitting the bug reports into four subsegments–each representing critical aspects of the report–TAB ensures that no vital information is overlooked during the title generation process. This segmentation allows the model to focus on specific, high-priority sections of the bug report, leading to more informative and concise titles. It also ensures that the generated titles reflect the most important details, enhancing the clarity and usefulness of the titles for bug tracking and prioritization.

These components collectively enable TAB to outperform existing methods like iTAPE by delivering more accurate and contextually appropriate titles, regardless of whether the bug reports are template-based or non-template-based. The robustness and adaptability of our approach ensure high performance across different types of bug report data, making TAB a superior tool for automated title generation.

6.3 The usage scenario of TAB

The primary usage scenario for TAB is as a plugin for bug report submission within bug tracking systems. For users preparing to submit a new bug report, TAB automatically generates a high-quality, concise title based on the content of the report. This reduces the burden on users to manually craft titles and ensures that submitted reports contain clear, informative titles from the start.

In addition to generating titles, TAB can also evaluate the reasonableness of existing bug report titles by comparing them to the content of the bug report. If a title does not adequately reflect the core issues or information presented in the report, TAB can suggest improvements or flag the title for revision. This feature helps ensure that all bug reports are consistently titled, contributing to the overall quality of issue tracking and communication.

By integrating TAB into bug-tracking systems, developers and project managers can ensure that all bug reports contain clear, concise, and informative titles, which can significantly improve the efficiency of issue triage and resolution. A well-structured title allows developers to quickly understand the core issue, aiding in quicker prioritization and assignment. This can be particularly valuable in large-scale projects with numerous contributors, where maintaining consistency in bug reporting is crucial for effective collaboration.

Application in Large Projects: TAB is particularly beneficial in large-scale software development environments, where there may be hundreds or thousands of bug reports generated by diverse contributors. The standardized titles produced by TAB can greatly enhance communication between teams by ensuring that each bug report is easy to understand at a glance. This improves collaboration across geographically dispersed teams and ensures that critical issues are addressed promptly.

Adaptability Across Domains: Although initially designed for bug report title generation, TAB's flexible architecture allows it to be adapted to various other software engineering tasks that require title or summary generation. For example, TAB could be applied to project management systems or documentation workflows where concise and informative summaries are crucial. Its integration into different domains highlights its versatility and potential to streamline content management and improve information accessibility across the software development lifecycle.

By ensuring clarity and consistency in bug report titles, TAB not only enhances issue tracking and resolution but also contributes to more organized, scalable, and effective bug management practices in large projects.

6.4 Threats to validity

One threat to validity is the evaluation metrics of issue title generation. To evaluate the performance of the title generation method, we employ three metrics(ROUGE-L, METEOR, chrF). Neither TAB nor baseline performs well enough on these evaluation metrics. Therefore, conclusions(i.e. our TAB performs better than baselines) drawn based on these metrics are not convincing enough.

The second threat to validity is that the dataset used for training and evaluation may not be fully representative of all possible bug report scenarios. While we have made efforts to collect a diverse and comprehensive dataset from GitHub repositories, there might still be variations in bug reporting practices across different projects and domains that are not captured in our dataset. This could potentially limit the generalizability of our findings.

Additionally, the performance of TAB might be influenced by the quality and structure of the input data. For example, non-template-based bug reports can vary

significantly in their format and content, making it challenging for the model to generate accurate titles consistently. Although our `Document Component Analyzer(DCA)` helps to mitigate this issue by organizing the input data, there is still a risk that the variability in non-template-based reports could impact the performance of TAB.

7 Related work

7.1 Analysis of bug report

Due to the widespread use of bug tracking systems by developers to discuss various issues in software development, Ko and Ko and Chilana (2011) conducted a qualitative analysis of the design discussions found in the complete set of closed bug reports from three open-source projects: Firefox, Linux kernel, and Facebook API. They discovered that many of the discussions centered around whether to adhere to the original design intentions or to adjust based on user needs. They also recommended redesigning online discussion tools to facilitate clearer suggestions. Sahoo etal. (2010) studied the reproducibility of bug reports by randomly sampling six server application reports. They found that in 77% of cases, a single request was sufficient to reproduce the bug. Xuan etal. (2012) proposed the first model that uses a sociotechnical approach to determine developer priorities within bug tracking systems. Specifically, through the analysis of tasks within the bug report repository, they concluded that establishing developer priorities helps enhance the handling of bug reports, particularly in the triage process. Bhattacharya etal. (2013) defined several metrics to assess the quality of Android bug reports. On the other hand, they compared Google's bug tracking system with Bugzilla and Jira, finding that although Google's bug tracker is more widely used in Android applications, it offers comparatively less management support. Wang and Zhang (2012) established a state transition model based on historical data and proposed a method to predict the number of bugs in various states of bug reports. This method can be used to forecast the future bug-fixing performance of a project.

7.2 Applications of bug report titles

Many studies have utilized bug report titles as a feature for analyzing bug reports. For instance, Sureka and Indukuri (2010) investigate the relationship between bug report titles and bug importance levels. Similarly, Tian etal. (2012) and Chaparro etal. (2019) identify duplicated bug reports using titles as one of their features. Additionally, Sun et al. (2017) and Ruan et al. (2019) focus on recovering the missing links between bug reports and commits based on their similarities, where bug report titles are regarded as important textual features. Other research also examines the impact of bug report titles on bug triaging (Chaparro etal. 2019) and Mills etal. (2018). These studies highlight the importance of improving the quality of bug report titles, as such enhancements could significantly benefit downstream research related to bug reports.

7.3 Generation task for bug reports

There are not many kinds of studies devoted to the generation task for bug reports. Most of the previous studies focused on extracting important sentences and generating summaries of bug reports. For example, Rastkar etal. (2010); Rastkar et al. (2014) proposed a conversion-based summarizer for bug reports by identifying important sentences of bug reports automatically. Jiang et al. (2017) summarize bug reports in consideration of the reporters' authorship. Mani etal. (2012) and Lotufo et al. (2015) proposed unsupervised bug report summarization approaches based on noise reducer or heuristic rules.

Besides bug report summarization tasks, the title generation for bug reports has become a new research direction. Chen etal. (2020) proposed an automatic method to generate titles for bug reports. They formulated title generation into a one-sentence summarization task. Different from their work, this work aims to generate titles for semi-structured bug reports which is a multi-sentence summarization task.

7.4 Other document generation for software artifacts

Prior studies have proposed diverse automated document generation approaches for software artifacts other than commit messages, such as code comments (Sridhara etal. 2010; Haiduc etal. 2010; Moreno etal. 2013; Wong etal. 2013; McBurney and McMillan 2014, 2016; Iyer etal. 2016; Hu etal. 2018, 2019; Wan etal. 2018; Zhang etal. 2020), release notes (Moreno etal. 2014; Moreno et al. 2016).

As for code comments generation, Hu etal. (2018, 2019) proposed an attentional encoder-decoder model-based approach to generate comments for Java methods. Wan etal. (2018) improved the encoder-decoder-based approach by using a hybrid encoder and a reinforcement learning-based decoder to generate code comments. Zhang etal. (2020) proposed a retrieval-based neural source code summarization approach that can take advantage of both neural and retrieval-based techniques.

As for release notes generation, Abebe et al. (2016) proposed a machine learning-based approach for automatically identifying the issues to be mentioned in release notes. Moreno etal. (2014); Moreno et al. (2016) proposed ARENA to generate release notes. ARENA first summarizes changes in a release and then integrates these summaries with their related information in the issue tracker.

These studies have inspired our work to generate titles for semi-structured bug reports to facilitate downstream tasks.

8 Conclusion

Writing high-quality bug report titles is crucial for efficient software development but remains a challenging task for many reporters. Although existing automated approaches for bug report title generation make progress, they often produce low-quality titles that can mislead developers and hinder the debugging process. In

this paper, we propose TAB, an automated framework designed to generate accurate and meaningful titles for bug reports. TAB is particularly effective for both template-based and non-template-based bug reports. For template-based reports, TAB directly generates titles by leveraging the structured information within the report. For non-template-based reports, TAB first applies a classification step to segment the report into meaningful components, then generates titles based on these classifications. We evaluate TAB on two datasets–one containing template-based bug reports and the other containing non-template-based reports–using several automatic metrics. The results demonstrate that TAB consistently outperforms existing approaches, showcasing its robustness and effectiveness in generating high-quality titles for a variety of bug report formats. This makes TAB a valuable tool for improving the accuracy and efficiency of bug triage in software development.

Acknowledgements This work was supported in part by the National Natural Science Foundation of China (No. 62372071), the Scientific and Technological Research Program of Chongqing Municipal Education Commission (No. KJQN202300547), the Chongqing Municipal Construction Science and Technology Plan Project (Chengke Zi 2024 No. 8-7), the State Key Laboratory of Intelligent Vehicle Safety Technology (No. IVSTSKL-202412) and the Natural Science Foundation of Chongqing (No. CSTB2023NSCQ-MSX0914).

References

Abebe, S.L., Ali, N., Hassan, A.E.: An empirical study of software release notes. Empir. Softw. Eng. **21**(3), 1107–1142 (2016)

Anonymous.: (2024). https://anonymous.4open.science/r/TAB-7E70/

Ba, J.L., Kiros, J.R., Hinton, G.E.: Layer normalization. arXiv preprint arXiv:1607.06450 (2016)

Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., Zimmermann, T.: What makes a good bug report? In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 308–318 (2008)

Bhattacharya, P., Ulanova, L., Neamtiu, I., Koduru, S.C.: An empirical analysis of bug reports and bug fixing in open source android apps. In: 2013 17th European Conference on Software Maintenance and Reengineering, pp. 133–143 (2013). https://doi.org/10.1109/CSMR.2013.23

Chaparro, O., Lu, J., Zampetti, F., Moreno, L., Di Penta, M., Marcus, A., Bavota, G., Ng, V.: Detecting missing information in bug descriptions. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 396–407 (2017)

Chaparro, O., Bernal-Cárdenas, C., Lu, J., Moran, K., Marcus, A., Di Penta, M., Poshyvanyk, D., Ng, V.: Assessing the quality of the steps to reproduce in bug reports. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 86–96 (2019)

Chaparro, O., Plorez, J.M., Singh, U., Marcus, A.: Reformulating queries for duplicate bug report detection. In: In Proceedings of The26th International Conference on Software Analysis,Evolution and Reengineering, pp. 218–229, IEEE (2019)

Chaparro, O., Plorez, J.M., Singh, U., Marcus, A.: Deeptriage:explor-ing the effectiveness of deep learning for bug triaging. In: In Proceedings of the Indiajoint International Conference on Data Science and Management of Data, pp. 171–179, Association for Computing Machinery (2019)

Chen, S., Xie, X., Yin, B., Ji, Y., Chen, L., Xu, B.: Stay professional and efficient: automatically generate titles for your bug reports. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 385–397, IEEE (2020)

Davies, S., Roper, M.: What's in a bug report? In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 1–10 (2014)

Devlin, J., Chang, M.-W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
Devlin, M.C., Lee, K., Toutanova, K.: Bert: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers), pp. 4171–4186, Association for Computational Linguistics (2019)
Fleiss, J.L.: Measuring nominal scale agreement among many raters. Psychol. Bull. **76**, 378–382 (1971)
Guo, S.L., N. Duan, Y.W., M. Zhou, J.Y.: Unixcoder: Unified cross-modal pre-training for code representation. In: in Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022, S. Muresan, P. Nakov, and A. Villavicencio, Eds, pp. 7212–7225, Association for Computational Linguistics (2022)
Haiduc, S., Aponte, J., Moreno, L., Marcus, A.: On the use of automated text summarization techniques for summarizing source code. In: Reverse Engineering (WCRE), 2010 17th Working Conference On, pp. 35–44, IEEE (2010)
Hu, X., Li, G., Xia, X., Lo, D., Jin, Z.: Deep code comment generation with hybrid lexical and syntactical information. Empir. Softw. Eng. **25**, 2179 (2019)
Hu, X., Li, G., Xia, X., Lo, D., Jin, Z.: Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension, pp. 200–210 (2018)
Huang, Q., Xia, X., Lo, D., Murphy, G.C.: Automating intention mining. IEEE Trans. Softw. Eng. **46**(10), 1098–1119 (2018)
Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), vol. 1, pp. 2073–2083 (2016)
Jiang, H., Zhang, J., Ma, H., Nazar, N., Ren, Z.: Mining authorship characteristics in bug repositories. Sci. China Inf. Sci. **60**(1), 1–16 (2017)
Karim, M.R., Ihara, A., Yang, X., Iida, H., Matsumoto, K.: Understanding key features of high-impact bug reports. In: 2017 8th International Workshop on Empirical Software Engineering in Practice (IWESEP), pp. 53–58, IEEE (2017)
Ko, A.J., Chilana, P.K.: Design, discussion, and dissent in open bug reports. In: Proceedings of the 2011 IConference. iConference '11, pp. 106–113. Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1940761.1940776
Ko, A.J., Myers, B.A., Chau, D.H.: A linguistic analysis of how people describe software problems. In: Visual Languages and Human-Centric Computing (VL/HCC'06), pp. 127–134, IEEE (2006)
Lavie, A., Agarwal, A.: Meteor: An automatic metric for mt evaluation with high levels of correlation with human judgments. In: Proceedings of the Second Workshop on Statistical Machine Translation. StatMT '07, pp. 228–231. Association for Computational Linguistics, USA (2007)
Lawrence, S., Giles, C.L., Tsoi, A.C., Back, A.D.: Face recognition: a convolutional neural-network approach. IEEE Trans. Neural Netw. **8**(1), 98–113 (1997)
Li, H., Yan, M., Sun, W., Liu, X., Wu, Y.: A first look at bug report templates on GitHub. J. Syst. Softw. **202**, 111709 (2023)
Lin, C.Y.: Rouge: A package for automatic evaluation of summaries. In: In Proceedings of the Workshop on Text Summarization Branches Out (WAS 2004) (2004)
Liu, P., Fu, J., Hayashi, H., et al.: Pre-train, prompt, and predict: a systematic survey of prompting methods in natural language processing. ACM Comput. Surv. **55**(9), 1–35 (2023)
Liu, Z., Xia, X., Hassan, A.E., Lo, D., Xing, Z., Wang, X.: Neural-machine-translation-based commit message generation: how far are we? In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 373–384 (2018)
Liu, Q., Liu, Z., Zhu, H., Fan, H., Du, B., Qian, Y.: Generating commit messages from diffs using pointer-generator network. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 299–309, IEEE (2019)
Lotufo, R., Malik, Z., Czarnecki, K.: Modelling the hurried bug report reading process to summarize bug reports. Empir. Softw. Eng. **20**(2), 516–548 (2015)
Mani, S., Catherine, R., Sinha, V.S., Dubey, A.: Ausum: approach for unsupervised bug report summarization. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, pp. 1–11 (2012)

McBurney, P.W., McMillan, C.: Automatic source code summarization of context for java methods. IEEE Trans. Softw. Eng. **42**(2), 103–119 (2016)

McBurney, P.W., McMillan, C.: Automatic documentation generation via source code summarization of method context. In: Proceedings of the 22nd International Conference on Program Comprehension, pp. 279–290, ACM (2014)

Mills, C., Pantiuchina, J., Parra, E., Bavota, G., Haiduc, S.: Are bug reports enough for text retrieval-based bug localiza-tion? In: In Proceedings of the International Conference on Software Maintenance and Evolution, pp. 381–392, IEEE (2018)

Moreno, L., Bavota, G., Di Penta, M., Oliveto, R., Marcus, A., Canfora, G.: Arena: an approach for the automated generation of release notes. IEEE Trans. Softw. Eng. **43**(2), 106–127 (2016)

Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L., Vijay-Shanker, K.: Automatic generation of natural language summaries for java classes. In: Program Comprehension (ICPC), 2013 IEEE 21st International Conference On, pp. 23–32, IEEE (2013)

Moreno, L., Bavota, G., Di Penta, M., Oliveto, R., Marcus, A., Canfora, G.: Automatic generation of release notes. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 484–495, ACM (2014)

Nijkamp, E., Pang, B., Hayashi, L. H. Tu, Wang, H., Zhou, Y., Savarese, S., Xiong, C.: Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474 (2022)

Popović, M.: chrf: character n-gram f-score for automatic MT evaluation. In: Proceedings of the Tenth Workshop on Statistical Machine Translation, pp. 392–395 (2015)

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J.: Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer (2023). https://arxiv.org/abs/1910.10683

Rastkar, S., Murphy, G.C., Murray, G.: Automatic summarization of bug reports. IEEE Trans. Softw. Eng. **40**(4), 366–380 (2014)

Rastkar, S., Murphy, G.C., Murray, G.: Summarizing software artifacts: a case study of bug reports. In: 2010 ACM/IEEE 32nd International Conference on Software Engineering, vol. 1, pp. 505–514, IEEE (2010)

Roy, D., Fakhoury, S., Arnaoudova, V.: Reassessing automatic evaluation metrics for code summarization tasks. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1105–1116 (2021)

Ruan, H., Chen, B., Peng, X., Zhao, W.: Deeplink: re-covering issue-commit links based on deep learning. J. Syst. Softw. **158**, 110406 (2019)

Sahoo, S.K., Criswell, J., Adve, V.: An empirical study of reported bugs in server software with implications for automated bug diagnosis. In: 2010 ACM/IEEE 32nd International Conference on Software Engineering, vol. 1, pp. 485–494 (2010). https://doi.org/10.1145/1806799.1806870

Sennrich, R., Haddow, B., Birch, A.: Neural machine translation of rare words with subword units. arXiv preprint arXiv:1508.07909 (2015)

Sharma, S., El Asri, L., Schulz, H., Zumer, J.: Relevance of unsupervised metrics in task-oriented dialogue for evaluating natural language generation. CoRR **abs/1706.09799** (2017)

Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., Vijay-Shanker, K.: Towards automatically generating summary comments for java methods. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 43–52, ACM (2010)

Sun, Y., Wang, Q., Yang, Y.: Frlink: improving the recovery of miss-ing issue-commit links by revisiting file relevance. Inf. Sofiw. Technol. **84**, 33–47 (2017)

Sureka, A., Indukuri, K.V.: Linguistic analysis of bugreport titles with respect to the dimension of bug importance. In: In Proceedings of the 3rd Annual Bangalore Conference, pp. 1–6, Association for Computing Machinery (2010)

Tabassum, J., Maddela, M., Xu, W., Ritter, A.: Code and named entity recognition in stackoverflow. arXiv preprint arXiv:2005.01634 (2020)

Tian, Y., Sun, C., Lo, D.: Improved duplicate bug re-port identification. In: In Proceedings of the 16th European Conference on SofiwareMaintenance and Reengineering, pp. 385–390, IEEE (2012)

Wan, Y., Zhao, Z., Yang, M., Xu, G., Ying, H., Wu, J., Yu, P.S.: Improving automatic source code summarization via deep reinforcement learning. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 397–407 (2018)

Wang, M.W., Y. Liu, Y.W., Shenyang, R.W.: Understanding and facilitating the co-evolution of production and test code. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 272–283, IEEE (2021)

Wang, J., Zhang, H.: Predicting defect numbers based on defect state transition models. In: Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 191–200 (2012). https://doi.org/10.1145/2372251.2372287

Wei, B.: Retrieve and refine: Exemplar-based neural comment generation. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1250–1252 (2019). https://doi.org/10.1109/ASE.2019.00152

Wong, E., Yang, J., Tan, L.: Autocomment: Mining question and answer sites for automatic comment generation. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference On, pp. 562–567, IEEE (2013)

Xuan, J., Jiang, H., Ren, Z., Zou, W.: Developer prioritization in bug repositories. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 25–35 (2012). https://doi.org/10.1109/ICSE.2012.6227209

Zhang, T., Chen, J., Luo, X., Li, T.: Bug reports for desktop software and mobile apps in GitHub: What's the difference? IEEE Softw. **36**(1), 63–71 (2017)

Zhang, J., Wang, X., Zhang, H., Sun, H., Liu, X.: Retrieval-based neural source code summarization. In: Proceedings of the 42nd International Conference on Software Engineering (2020)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Authors and Affiliations

Xiao Liu[1] · Yinkang Xu[1] · Weifeng Sun[1] · Naiqi Huang[1] · Song Sun[2] · Qiang Li[1] · Dan Yang[3] · Meng Yan[1]

✉ Weifeng Sun
weifeng.sun@cqu.edu.cn

✉ Meng Yan
mengy@cqu.edu.cn

Xiao Liu
cdjx@cqu.edu.cn

Yinkang Xu
xuyinkang@stu.cqu.edu.cn

Naiqi Huang
npcxh@cqu.edu.cn

Song Sun
20220033@cqnu.edu.cn

Qiang Li
liqiang@stu.cqu.edu.cn

Dan Yang
dyang@cqu.edu.cn

[1] School of Big Data and Software Engineering, Chongqing University, Chongqing, China

[2] Chongqing Normal University, Chongqing, China

[3] Southwest Jiaotong University, Chengdu, China