CAMCMG: A Change-aware Model for Commit Message Generation

Anonymous ACL submission

Abstract

Commit messages play a crucial role in understanding code change intent and tracking software evolution, and automatically generating high-quality commit messages is important for improving software maintenance efficiency. Although existing methods have made preliminary attempts to capture the semantics of *diff*, semantic associations across hunks and the semantic alignment between the *diff* and its commit message have not been further explored. In this paper, we propose a change-aware model for commit message generation called CAMCMG, which focuses on the hunk associations in code change and alignment between code change and commit message. Specifically, it adopts a local attention mechanism based on a sliding window to capture local contextual information around hunks, a CMG-oriented change attention mechanism to model semantic associations across hunks, and a change alignment loss to optimize the model to enhance the generation of change-aligned tokens during the training phase. Experimental results show that CAMCMG achieves an average improvement of 6.4% on automatic evaluation metrics and 5.2% on human evaluation metrics Ablation studies further demonstrate the effectiveness of each component, where local attention, change attention, and change alignment loss contribute average improvements of 5.1%, 9.6%, and 7.1%, respectively, on the automatic evaluation metrics.

1 Introduction

Commit message summarizes the purpose and intent of code change[1], which plays an important role in program comprehension, such as code review (Caulo et al., 2020; Li et al., 2025), defect localization (Wang et al., 2025; Li et al., 2021; Schlüter et al., 2021) and software evolution analysis (Gîrba and Ducasse, 2006; Wei et al., 2025).

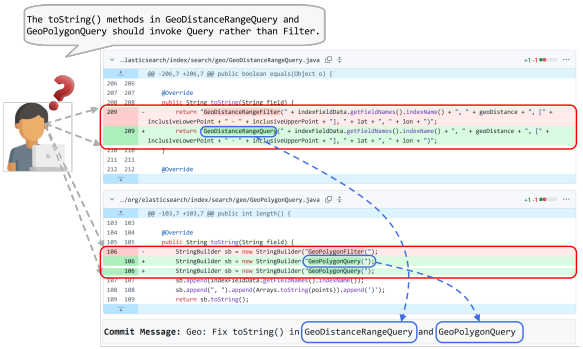

Figure 1: Two hunks modify the code by changing the suffix of `GeoPolygonFilter` and `GeoDistanceRangeFilter` to `Query`. The commit message summarizes the modified methods and classes reflected in the diff.

High-quality commit messages can help developers quickly understand the context of changes and improve the efficiency of team collaboration. However, there are many commit messages that are vague or even blank in real-world projects (Maalej and Happel, 2010), since writing commit messages manually is time-consuming and tedious (Zhang et al., 2024c). For example, approximately 14% of the commit messages in over 23,000 SourceForge projects were completely empty (Dyer et al., 2013); and this problem also existed in the Apache Software Foundation projects (Imani et al., 2024). Therefore, various automatic commit message generation (CMG) methods have been proposed to generate accurate, concise, and semantically logical descriptions for code changes. Specifically, given a code change $diff = \{h_1, h_2, \ldots, h_n\}$, where h_i denotes the i-th hunk, CMG aims to transform *diff* to a commit message.

Existing CMG methods can be broadly classifed into template-based (Buse and Weimer, 2010; Shen et al., 2016), retrieval-based (Huang et al., 2020; Liu et al., 2018; Hoang et al., 2020), learning-based (Dong et al., 2022; Liu et al., 2023; Nie

[1]In this paper, we use *diff* and *code change* exchangeably.

et al., 2021; Wang et al., 2024a) and hybrid methods (Liu et al., 2020; Shi et al., 2022; Zhang et al., 2024b; He et al., 2023). Recently, with the rising of pre-trained language models (PLMs), PLM-based methods become dominant in learning-based methods. These methods are based on the Transformer architecture, which adopts the standard full attention mechanism (Vaswani et al., 2017). However, standard full attention mechanism computes the attention scores among arbitrary tokens, resulting in the attention being diffusely distributed (Correia et al., 2019). Many studies attempt to leverage a sparse attention mechanism to mitigate this issue (Beltagy et al., 2020; Guo et al., 2023; Wang et al., 2024b). Despite their efforts, there is still a gap between the CMG scenario and the general scenario. The following two key observations provide an opportunity to design a CMG-oriented sparse attention mechanism.

(O1): Semantic associations among hunks. Even if the different hunks in a *diff* are positionally separated, they are not completely semantically independent. As shown in Figure 1, the *diff* modifies the `toString()` method. Although the two hunks are separate in location, they share the same intent, i.e., calling class `Query` instead of `Filter`. Therefore, modeling cross-hunk semantic associations is necessary to capture the intent of code changes.

(O2): Semantic alignment between *diff* and commit message. Tokens in the commit message often correspond to certain changes in the *diff* (e.g., function calls, variable usage). As shown in Figure 1, the commit message summarizes the changes in *diff*, which involve the `toString()` method in both the `"GeoDistanceRangeQuery"` and `"GeoPolygonQuery"` classes. It emphasizes the importance of aligning the semantic content between the *diff* and its commit message.

In summary, generating high-quality commit messages requires modeling semantic associations across multiple hunks and aligning the generated message with the *diff*. However, to the best of our knowledge, *off-the-peg* methods do not sufficiently satisfy these *desiderata*. Therefore, this leads to the major challenges: how to associate the semantics across hunks and how to align the *diff* and its corresponding commit message.

Motivated by this, we propose CAMCMG, which is a Change-aware Model for Commit Message Generation. First, context plays an important role in commit message generation because the effectiveness of the generated message depends on the surrounding context, except for code changes (Sillito et al., 2008). Therefore, we employ a sliding window to capture local contextual information around each hunk. It allows each token to pay attention to the neighboring tokens. Second, we further introduce a CMG-oriented change attention mechanism to capture semantic associations across hunks, which enables related tokens among different hunks to establish attention connections. Finally, we design a change alignment loss to optimize the model to enhance the generation of tokens in the commit message that correspond to the changes of *diff* (denoted as change-aligned tokens) during the training phase.

To evaluate the effectiveness of CAMCMG, we conduct extensive experiments on MCMD, one of the most widely used datasets for CMG, and compare it with ten state-of-the-art (SOTA) methods. Experimental results show that CAMCMG achieves the best overall performance, with an average improvement of 6.4% on automatic evaluation metrics and 5.2% on human evaluation metrics compared with baseline models except for GPT-4. Ablation studies further confirm the effectiveness of each component. Specifically, local attention, change attention, and change alignment loss contribute average improvements of 5.1%, 9.6%, and 7.1%, respectively, on the automatic evaluation metrics. The main contributions of our paper can be summarized as follows:

- We propose CAMCMG, which includes local attention and change attention. Specifically, *local attention* focuses on capturing the local contextual information around each hunk, while *change attention* is designed specifically for the CMG task to model the semantic associations across hunks.

- We design a change alignment loss to optimize the model to enhance the generation of change-aligned tokens during the training phase, which directs the model to pay more attention to the semantic alignment tokens, thus improving the relevance and accuracy of commit messages.

- Extensive experiments demonstrate that CAMCMG outperforms the state-of-the-art methods on multiple evaluation metrics, which show its effectiveness and superiority.

2 Related Work

2.1 Commit Message Generation

Researchers have proposed various methods to generate commit messages. Specifically, it can be categorized as templated-based, retrieval-based, learning-based, hybrid and LLM-based methods. Early work generated commit messages based on predefined templates (Buse and Weimer, 2010; Shen et al., 2016). For example, Buse et al. (Buse and Weimer, 2010) leveraged control flow patterns to generete commit messages. Retrieval-based methods retrieved similar commits from the training set and reused their commit messages (Liu et al., 2018; Huang et al., 2020; Hoang et al., 2020). For example, Liu et al. (Liu et al., 2018) used cosine similarity to retrieve similar code changes. Learning-based methods adopted neural network models to translate *diffs* into commit messages. For example, Dong et al. (Dong et al., 2022) proposed FIRA, which builds fine-grained graphs by combining sub-tokens nodes with edit operation nodes to capture small differences in code changes. Liu et al. (Liu et al., 2023) proposed CCRep, which uses a pre-trained model with a query-back mechanism to model interactions between code changes and their contexts. Hybrid methods leverage both retrieval-based and learning-based approaches to generate commit messages. For example, ATOM (Liu et al., 2020) extracts semantic information from code changes using ASTs and combines retrieval and generation through hybrid ranking. RACE (Shi et al., 2022) retrieves similar code–message pairs to guide generation. COME (He et al., 2023) learns contextualized code change representations via modification embeddings and a self-supervised objective, and integrates retrieval and generation through a decision algorithm. Recent work has explored LLMs for commit message generation. Wu et al. (Wu et al., 2025) empirically validated the effectiveness and generalizability of LLMs through in-context learning.

Although the above methods have achieved promising performance, the semantic alignment between *diffs* and commit messages is rarely modeled explicitly, which potentially affects the accuracy and expressiveness of the resulting messages.

2.2 Sparse Attention Mechanisms

The self-attention mechanism allows each token to attend to all others (Zaheer et al., 2020). In long code sequences, this indiscriminate attention can dilute focus on semantically related tokens across distant hunks, reducing the model's ability to capture code semantics. To mitigate this, prior work has explored sparse attention mechanisms. For example, Wang et al. (Wang et al., 2024b) proposed SparseCoder, which modeled short-term dependencies using a sliding-window mechanism and introduced global attention and identifier attention to capture long-range dependencies among source code identifiers. Guo et al. (Guo et al., 2023) proposed LongCoder, which introduced bridge attention and global attention, effectively captured long-range dependency while maintaining fine-grained local semantic representations. Zhang et al. (Zhang et al., 2024a) proposed VulAdvisor, which incorporated local attention that used TF-IDF weights to emphasize locally important code elements and introduced repair action loss that prioritized semantically meaningful tokens during training, improving the actionability and precision of the suggestions.

Although existing sparse attention methods effectively address the challenges of long sequences and capture long-range dependencies in code, they generally lack explicit modeling of the semantic relationships among different hunks within a *diff*. Consequently, cross-hunk associations that are crucial for representing the overall semantics of code changes cannot be adequately captured.

3 Methodology

The overall framework of CAMCMG is illustrated in Figure 2. Given a *diff* as input, the first step is to construct specialized mask matrices for local attention and change attention. These mask matrices serve to indicate the key regions the model should focus on, enabling the model to focus on both the local context of each hunk and the semantic associations across multiple hunks. Once the mask matrices are prepared, they are fused by taking the element-wise maximum of them. Subsequently, it is integrated into the Transformer's self-attention mechanism. This allows the model to capture the local contextual information around hunks and associations among hunks. Formally, the standard self-attention is computed as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V, \quad (1)$$

$$Q = XW^Q \;\; K = XW^K \;\; V = XW^V, \quad (2)$$

where X is the embedded representation of the input sequence, and W^Q, W^K, and W^V are learn-

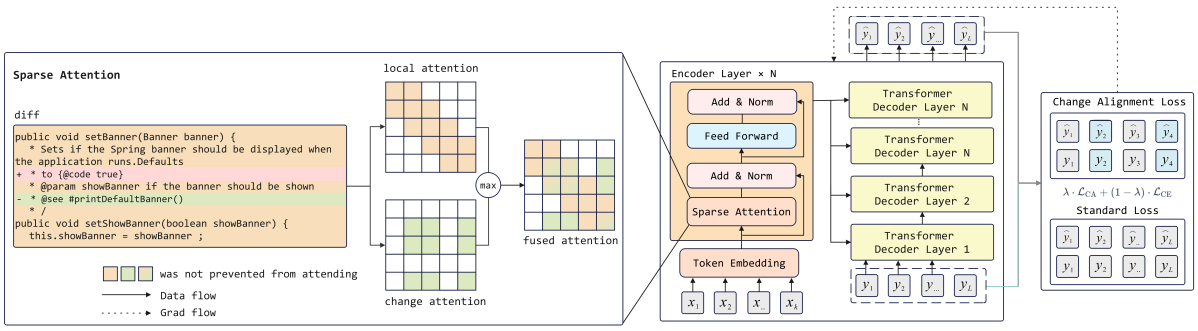

Figure 2: The overall workflow of CAMCMG.

able weight matrices. With the mask matrix incorporated, the attention computation is modified as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V \tag{3}$$

where M denotes the mask matrix, which restricts the range of attention computation, thereby controlling the positions each token can attend to when constructing contextual representations. If the i-th token is allowed to attend to the j-th token, $M_{ij} = 0$; otherwise $M_{ij} = -\infty$. When M_{ij} is set to $-\infty$, the attention score from the i-th to j-th token will be zero after using the softmax function.

During the training phase, the model is further optimized with a change alignment loss, which encourages the generated message to semantically align with the key areas captured in the *diff*.

3.1 Local Attention

To enable the model to effectively capture the local contextual information around each hunk, we introduce a local attention mechanism. Unlike the self-attention mechanism in Transformers, local attention does not allow each token to attend to all others; instead, it restricts the attention scope to a limited neighborhood. This design enables the model to learn fine-grained contextual dependencies while reducing the influence of irrelevant regions. Given a local window size ω, we define a range of local attention for each token in the *diff*, which only includes $\lfloor \omega/2 \rfloor$ tokens before and after the current token. The local attention mask M^{local} is thus constructed as:

$$M_{ij}^{\text{local}} = \begin{cases} 0 & \text{if } |i-j| \leq \lfloor \omega/2 \rfloor \\ -\infty & \text{otherwise,} \end{cases} \tag{4}$$

where i and j denote the positions of tokens in the *diff* sequence.

3.2 Change Attention

The *diff* typically consists of multiple hunks, each describing a localized semantic modification. However, real-world commits often involve multiple interrelated changes that may occur in different functions, classes, or modules, leading to semantic associations that span across hunks. Although local attention can effectively capture contextual information around a hunk, it fails to model the semantic linkages across hunks, which are essential for understanding the overall intent of a code change. For example, as shown in Figure 1, the `toString()` methods in two different files were modified, where the class names in the output strings were changed from "GeoDistanceRangeFilter" and "GeoPolygonFilter" to their corresponding "GeoDistanceRangeQuery" and "GeoPolygonQuery".

Therefore, we introduce change attention, which explicitly captures semantic relations across different hunks. We identify all hunks in the *diff* and represent their token position sets as $\mathcal{C}$. Tokens from different hunks can attend to each other within $\mathcal{C}$, allowing semantic information to flow across hunks. This enables the model to aggregate related modification cues and form a unified understanding of the overall code change. The change attention mask M^{change} is thus constructed as:

$$M_{ij}^{\text{change}} = \begin{cases} 0, & \text{if } i, j \in \mathcal{C} \\ -\infty, & \text{otherwise.} \end{cases} \tag{5}$$

The above two attention mechanisms enable CAMCMG to capture and integrate semantic associations around and across hunks, constructing a more comprehensive representation of the entire code change.

The mask matrix M in Eq.(1) is defined as:

$$M = \max\left(M_{ij}^{\text{local}},\, M_{ij}^{\text{change}}\right), \tag{6}$$

where $\max(\cdot,\cdot)$ is the element-wise maximum function.

3.3 Change Alignment Loss

In practice, certain tokens in the diff (such as variable names, function names, or class names) often carry crucial semantic information that is explicitly reflected in the commit message. For example, when developers rename a function or variable, or add a new method, the commit message typically describes these specific changes. This indicates that there exists a semantic alignment between tokens in the diff and the commit message, which is important for accurately generating commit messages. Therefore, during model optimization, we introduce a change alignment loss, which builds upon the standard cross-entropy loss to further enhance the model's focus on the actual modifications in the *diff*. The model is able to more accurately capture the core modification information and achieve semantic alignment between the *diff* and its commit message by applying additional optimization constraints to the change-aligned tokens.

Algorithm 1: Change Alignment Mask Generation

Input: Change code $diff$, target commit message cm_t, generated message tokens t_{cm_g}
Output: Change aligment mask ca_mask.

1 $W_{diff} \leftarrow$ tokenize $diff$ by spaces and lemmatize them
2 $W_{cm_t} \leftarrow$ tokenize cm_t by spaces and lemmatize them
3 $W_{co} \leftarrow W_{diff} \cap W_{cm_t}$
4 $cm_g = decode(t_{cm_g})$ ▷ decode tokens to word sequence
5 $ca\_mask \leftarrow [0] \times |t_{cm_g}|$
6 **for** w ***in*** cm_g **do**
7 $w \leftarrow$ lemmatize w
8 **if** $w \in W_{co}$ **then**
9 set the elements in ca_mask whose index is equal to the index of w in t_{cm_g} to 1
10 **Return** ca_mask

Algorithm 1 outlines the process of identifying the change-aligned tokens and creating their masks. The algorithm first tokenizes and lemmatizes both the *diff* and the target commit message to obtain their respective word sets (Lines 1–2). In Line 3, it identifies the words that appear in both the changed regions of the *diff* and the target commit message, extracting their intersection. Subsequently, the generated message is decoded into a complete word sequence, and a zero mask matrix with the same expected length is initialized (Lines 4–5). Then, for each word in the generated message, the algorithm checks whether its lemmatized form exists in the intersection set. If the word appears in the intersection, the corresponding position in the mask is set to 1, indicating the parts of the generated message that are semantically aligned with the change regions (Lines 6–9). Through this process, the algorithm creates a mask ('ca_mask') that highlights the alignment positions between the generated message and the changed code.

Next, we incorporate the change alignment mask into the loss function, guiding the model to focus on generating these important tokens correctly. The change alignment loss is computed as

$$\mathcal{L}_{CA} = -\sum_{i=1}^{L} \text{ca\_mask}_i \, \log P(y_i \mid y_{<i}), \tag{7}$$

where L denotes the length of the generated sequence, and ca_mask_i is the value of the change alignment mask for the i-th token, corresponding to the probability of generating the i-th token in the sequence.

The final loss $\mathcal{L}_{\text{final}}$ is defined as the weighted sum of the change-aligned loss ($\mathcal{L}_{CA}$) and the standard cross-entropy loss ($\mathcal{L}_{CE}$):

$$\mathcal{L}_{\text{final}} = \lambda \cdot \mathcal{L}_{CA} + (1-\lambda) \cdot \mathcal{L}_{CE}, \tag{8}$$

where λ is a hyperparameter controlling the balance between the two losses.

4 Evaluation

In this section, we conduct comprehensive experiments to evaluate the effectiveness of CAMCMG, aiming to address the following research questions.

- RQ1: **Overall performance**. How does CAMCMG perform compared to the state-of-the-art commit message generation methods?

Method	Java					C#					C++					Python					JavaScript				
	BLEU	Met.	Rou.	Cid.	BRSA	BLEU	Met.	Rou.	Cid.	BRSA	BLEU	Met.	Rou.	Cid.	BRSA	BLEU	Met.	Rou.	Cid.	BRSA	BLEU	Met.	Rou.	Cid.	BRSA
NNGen (Liu et al., 2018)	19.41	12.40	25.15	1.23	29.40	20.93	13.52	25.14	1.43	29.42	12.72	8.48	16.85	0.67	21.71	15.62	10.42	20.97	0.88	25.46	17.96	11.88	23.47	1.15	27.78
RACE (Shi et al., 2022)	25.66	15.46	32.02	1.76	37.98	26.33	16.37	31.31	1.84	36.81	19.13	12.55	24.52	1.14	31.07	21.79	14.68	28.35	1.40	34.45	25.55	16.31	31.79	1.84	37.34
COME (He et al., 2023)	**27.17**	16.91	34.59	1.90	41.13	**27.29**	**17.77**	33.33	**1.91**	38.96	20.80	14.55	27.01	1.25	33.93	**23.17**	16.46	30.48	1.50	37.06	26.91	17.84	34.44	1.92	39.93
CodeT5-base (Wang et al., 2021b)	22.76	14.57	30.23	1.43	36.42	22.21	14.51	29.08	1.33	34.22	16.73	11.69	22.86	0.85	29.85	17.99	12.74	25.27	0.96	31.67	22.87	15.12	29.81	1.50	34.76
CCT5 (Lin et al., 2023)	26.46	16.58	33.95	1.81	40.81	26.89	17.20	32.67	1.86	38.84	19.80	13.82	26.11	1.15	33.37	22.54	15.77	29.92	1.44	36.57	26.16	17.17	33.44	1.86	39.12
Claude3 (The)	7.69	5.47	9.33	0.21	21.97	7.01	4.92	8.49	0.19	19.99	7.23	5.78	8.52	0.19	20.79	7.56	6.14	8.61	0.19	20.85	7.65	5.29	8.96	0.19	20.34
DeepSeekV2 (Liu et al., 2024)	6.56	4.88	7.96	0.21	21.47	5.94	4.25	6.96	0.18	19.03	6.47	5.13	7.27	0.19	19.95	6.84	5.45	7.41	0.20	20.05	6.41	4.65	7.24	0.19	19.34
Llama3 (Dubey et al., 2024)	7.88	6.29	9.63	0.25	24.13	7.25	5.69	8.58	0.21	21.58	7.97	7.08	8.95	0.24	22.86	8.43	7.43	9.15	0.25	22.71	7.64	5.97	8.99	0.23	21.78
CodeQwen1.5 (Bai et al., 2023)	7.37	6.49	8.93	0.21	22.54	7.09	5.81	7.99	0.17	20.51	7.56	6.74	8.34	0.19	21.38	7.93	7.01	8.18	0.20	21.11	6.87	6.08	8.01	0.18	20.06
GPT-4 (Achiam et al., 2023)	9.41	7.32	12.03	0.35	25.35	8.50	7.33	11.66	0.28	23.83	8.84	8.48	10.82	0.27	25.79	9.85	8.94	11.18	0.31	24.86	8.94	7.98	12.41	0.32	26.03
CAMCMG (Ours)	27.10	**17.63**	**38.30**	**2.06**	**41.90**	25.38	17.17	**35.70**	1.88	**39.31**	**20.91**	**14.83**	**31.50**	**1.57**	**35.07**	21.31	**16.87**	**35.54**	**1.59**	**37.93**	**29.02**	**20.68**	**41.13**	**2.28**	**43.84**

Table 1: The overall performance of CAMCMG. The best score is highlighted in **bold** and the second best score is underlined.

- RQ2: **Ablation study**. How does each component of CAMCMG contribute to the effectiveness?
- RQ3: **Human evaluation**. How does CAMCMG perform from the perspective of developers?

4.1 Evaluation Criteria

We evaluate the quality of the generated messages using five metrics: BLEU (Papineni et al., 2002)(the BLEU results reported in this paper are B-Norm), ROUGE (Lin, 2004), METEOR (Banerjee and Lavie, 2005), CIDEr (Vedantam et al., 2015) and BRSA (Li et al., 2024). The detailed metrics calculation can be found in Appendix A.

4.2 Dataset

The MCMD dataset, collected by Tao et al. (Tao et al., 2022), includes five programming languages: Java, C++, C#, Python, and JavaScript, and has been widely used in recent years for commit message generation (He et al., 2023; Shi et al., 2022; Tao et al., 2024). In this paper, we use the same version of the dataset as that used in COME (He et al., 2023) for our experiments.

4.3 Evaluation Models

We compare CAMCMG with one retrieval method (NNGen (Liu et al., 2018)), two learning-based methods (CodeT5 (Wang et al., 2021b), CCT5 (Lin et al., 2023)), two hybrid methods (COME (He et al., 2023), RACE (Shi et al., 2022)) and five LLM-based methods (Claude3-Haiku (The), DeepSeek-V2-16B (Liu et al., 2024), Llama3.1-8B (Dubey et al., 2024), CodeQwen1.5-7B (Bai et al., 2023), GPT-4 (Achiam et al., 2023)).

4.4 Experimental Setting

For the dataset, we set the maximum sequence length of *diff* to 512 tokens and commit messages to 50 tokens. We initialize our model using the pre-trained CodeT5-base while training the encoder-decoder neural network. The original CodeT5 vocabulary contains 32,100 tokens, and following Shi et al. (Shi et al., 2022), we extend it with nine special tokens (<INSERT>, <INSERT_OLD>, <DELETE>, <DELETE_END>, <KEEP>, <KEEP_END>,<REPLACE_OLD>, <REPLACE_OLD>, <REPLACE_END>), resulting in a vocabulary size of 32,109. We use the AdamW optimizer with a learning rate of 5e-5, and the batch size is 16. The maximum number of epochs is set to 10. All experiments are conducted on an NVIDIA GeForce RTX 4090 GPU. For GPT-4, following prior works (Lopes et al., 2024; Wu et al., 2025), we randomly select 200 samples from the test set to reduce the number of API calls.

5 Results And Aanlysis

In this section, we first present the overall performance of CAMCMG (RQ1) in Section 5.1, followed by the results of the ablation study (RQ2) in Section 5.2, and then discuss the results of human evaluation (RQ3) in Section 5.3.

5.1 RQ1: Overall Performance

Table 1 presents the experimental results of different models on the MCMD dataset. As shown in Table 1, CAMCMG achieves the best performance, with an average improvement of 6.4% over the baselines. Specifically, METEOR improves by 4.2% on average, ROUGE-L by 14.1%, CIDEr by 11.4%, and BRSA by 3.7%. However, the BLEU score decreases by 1.4% on average. This is because BLEU relies on n-gram precision, so outputs that are semantically correct but lexically different may receive low scores (Wieting et al., 2019). In contrast, BRSA which measures semantic similarity remains higher, showing that the generated

Method	Java					C#					C++					Python					JavaScript				
	BLEU	Met.	Rou.	Cid.	BRSA	BLEU	Met.	Rou.	Cid.	BRSA	BLEU	Met.	Rou.	Cid.	BRSA	BLEU	Met.	Rou.	Cid.	BRSA	BLEU	Met.	Rou.	Cid.	BRSA
- LA	24.72	17.10	36.94	1.92	41.04	23.75	16.25	33.87	1.62	37.56	19.07	14.63	29.78	1.32	33.46	19.92	15.96	33.70	1.52	37.37	28.60	20.02	40.54	2.17	42.91
- CA	24.06	15.88	34.44	1.73	37.72	22.57	15.02	32.17	1.57	35.41	19.44	13.42	28.46	1.38	31.90	19.96	14.54	31.51	1.43	34.92	27.93	19.23	39.15	2.13	42.24
- CAL	26.04	16.44	36.39	1.89	39.81	22.68	15.12	32.19	1.58	35.70	19.79	13.96	29.26	1.41	32.77	20.98	15.34	32.91	1.50	34.80	28.03	19.38	38.96	2.16	42.20

Table 2: Results of the ablation study across different programming languages. LA, CA, and CAL denote Local Attention, Change Attention, and Change Alignment Loss, respectively.Like "-LA" indicate that the corresponding component has been removed in the ablation study.

messages are still semantically accurate despite the low BLEU.

It is worth noting that although LLM-based methods achieve relatively lower scores on automatic evaluation metrics, this does not necessarily indicate poor generation quality. Automatic evaluation metrics can measure the textual differences between generated messages and reference messages, but they are limited in capturing semantic consistency. Commit messages often have multiple expressions, and a generated message may use different words from the reference while conveying the same meaning, which can lead to underestimation by automatic metrics (Wang et al., 2024a). Therefore, to more accurately assess the quality of generated commit messages, we conducted a human evaluation in Section 5.3.

5.2 RQ2: Ablation Study

We conducted additional experiments to analyze the impact of different components in CAMCMG on the overall performance. Table 2 presents the performance comparison between the complete CAMCMG model and its ablated variants across different programming languages.

First, to investigate the impact of different attention mechanisms on the overall performance, we remove the local attention and the change attention separately. As shown in Table 2, when local and change attention are removed, the model's average performance drops by approximately 5.3% and 9.6%, respectively. This indicates that both attention mechanisms positively contribute to the model's performance, and the change attention has a more pronounced impact.

We further conduct experiments to explore the impact of the change alignment loss on the overall performance of CAMCMG. As shown in the last row of Table 2, removing this component leads to an average performance drop of approximately 7.1%. Specifically, BLEU decreases by 5.0%, METEOR by 8.0%, ROUGE-L by 6.9%, CIDEr by 9.1%, and BRSA by 6.5%. It indicates that the change alignment loss plays a positive role in improving model performance.

In summary, the results clearly show that removing any of these components leads to a performance drop, indicating that each part contributes meaningfully to the model's overall performance. In particular, the change attention mechanism yields the most substantial improvement, highlighting its importance in capturing cross-hunk semantic associations critical for commit message generation.

5.3 RQ3: Human Evaluation

Method	Informativeness	Conciseness	Expressiveness
NNGen	2.72	2.26	2.68
RACE	2.85	2.46	2.76
COME	2.87	2.59	2.83
CodeT5-base	2.85	2.54	2.84
CCT5	2.92	2.67	2.82
Claude3	3.17	3.03	3.01
DeepSeekV2	3.02	3.14	3.07
Llama3	3.10	2.98	3.12
CodeQwen1.5	2.99	3.05	3.14
GPT-4	**3.34**	**3.29**	**3.38**
CAMCMG (Ours)	3.28	3.24	3.19

Table 3: The result of human evaluation.

Following previous work (Liu et al., 2020; Wang et al., 2021a), we randomly select 100 commits from the testing set and design a questionnaire for manual evaluation. Each question in the questionnaire consisted of a code change, the corresponding reference message, and messages generated by various baseline models (The example of questionnaire can be seen in Appendix C). The study invited 2 PhD students and 3 Master students from the computer laboratories to participate in the survey, all of whom have at least 3 years of programming experience. Participants were asked to score each generated message on a scale of 1-5 across three dimensions: **Informativeness** (the extent to which the message reflects important information about the *diff*), **Conciseness** (the brevity and focus of the message), and **Expressiveness** (the clarity and naturalness in conveying the intent of the *diff*). The

performance of each model was measured by the average scores across all generated messages.

Table 3 presents the results of human evaluation. The results show that CAMCMG achieves the second-best overall performance, ranking slightly below GPT-4, while outperforming the best non-GPT baseline by 5.2%. Although GPT-4 attains higher scores, it incurs substantially higher costs in practical applications. Overall, CAMCMG can generate high-quality commit messages while maintaining lower usage costs. In addition, we conducted Wilcoxon signed-rank tests (Wilcoxon et al., 1963) on the human evaluation results to examine the significance of the differences between CAMCMG and other baselines. The results indicate that the performance differences are statistically significant, with all p-values smaller than 0.05 at the 95% confidence level.

6 Case Study

To further demonstrate the performance of our model in real scenarios, we analyze two cases and compare the generated messages with baseline methods.

```
@@ -521,7 +521,7 @@ public String getCode() {
    final int getCodeLength() {
-       return content() == null ? getCode().length() : content().getCodeLength();
+       return content().getCodeLength();
    }
@@ -748,7 +748,7 @@ public final LineLocation createLineLocation(int lineNumber) {
     * different source types.
     */
    Object getHashKey() {
-       return content() == null ? getName() : content().getHashKey();
+       return content().getHashKey();
    }
    final TextMap getTextMap() {
```

```
Commit message:
 Reference:    Always delegate to content() as it has to be always non-null
 NNGen:        Source . newFromTest is supported
 RACE:         merge pull request in g / truffle from ~ michael . van . de . vanter_oracle . com
 COME:         merge pull request in g / truffle from paw / fix - source - equals to master
 CodeT5-base:  Merge pull request in G / truffle from bugfix / SourceImpl to master
 CCT5:         each instances of source needs its content ( ) .
 Claude3:      Refactor Source class to delegate equals and hashCode to Content
 DeepSeekV2:   Updated getCode and getHashKey methods to use newContent method
 Llama3:       Refactor Source.java to use content().equals() and content().hashCode()
 CodeQwen:     Refactor and optimize Source class for better performance and readability.
 GPT-4:        Simplify Source methods by removing unnecessary null handling for content()
 CAMCMG:       remove redundant null checks for content() in Source.equals() and hashCode()
```

Figure 3: Example of removing redundant null checks.

Figure 3 presents a real-world case (the full diff can be found on Appendix E), which includes the code changes, the reference commit message, as well as the commit messages generated by CAMCMG and other baseline models. In this case, the developers refactored the `Source` class by removing redundant null checks related to the `content()` method. The manually written commit message accurately captures the developers' intent. The diff snippet shown in the figure contains multiple hunks across different functions, all aimed at removing unnecessary null checks for `content()`. From the generated results, only CAMCMG and Llama3 produce commit messages that are largely consistent with the reference message. In contrast, the outputs from other baseline models show varying degrees of deviation from the expected results.

```
@@ -263,6 +263,9 @@ public Message deserialize(JsonElement json, Type typeOfT,
          Message.Builder protoBuilder =
              (Message.Builder) getCachedMethod(protoClass, "newBuilder").invoke(null);
+         Message defaultInstance =
+             (Message) getCachedMethod(protoClass, "getDefaultInstance").invoke(null);
+
          Descriptor protoDescriptor =
              (Descriptor) getCachedMethod(protoClass, "getDescriptor").invoke(null);
          // Call setters on all of the available fields
@@ -300,8 +303,7 @@ public Message deserialize(JsonElement json, Type typeOfT,
                fieldValue = context.deserialize(jsonElement, protoArrayFieldType);
                protoBuilder.setField(fieldDescriptor, fieldValue);
              } else {
-               Message prototype = protoBuilder.build();
-               Object field = prototype.getField(fieldDescriptor);
+               Object field = defaultInstance.getField(fieldDescriptor);
                fieldValue = context.deserialize(jsonElement, field.getClass());
                protoBuilder.setField(fieldDescriptor, fieldValue);
              }
```

```
Commit message:
  Reference:    avoid repeated calls to build() during deserialization by obtaining field types
                from the Message's defaultInstance
  NNGen:        API Change : Implemented various getAs methods to return null .
  RACE:         allow subclasses to provide their own default instance .
  COME:         use a cached default instance of the proto type adapter .
  CodeT5-base:  Use getDefaultInstance ( ) instead of newBuilder ( ) .
  CCT5:         fix deserialization of default instance messages .
  Claude3:      Add support for Message.setField with default instance and prototype
  DeepSeekV2:   Update ProtoTypeAdapter to use getDefaultInstance and setters for deserialization
  Llama3:       Improved ProtoTypeAdapter to handle null fields and default instances
  CodeQwen:     Implement a custom Gson type adapter for Protobuf messages.
  GPT-4:        Optimize deserialization by using defaultInstance instead of building a prototype
                instance for field access
  CAMCMG:       Use defaultInstance instead of building a new prototype in deserialization
```

Figure 4: Example of instance reuse in deserialization.

To further verify the model's capability in understanding semantic changes in code, we conducted a case study on a real-world commit related to "protobuf message deserialization optimization". As shown in Figure 4, this commit replaces the invocation of `protoBuilder.build()` with `getDefaultInstance()`, thereby avoiding repeated construction of temporary prototype objects during deserialization and improving execution efficiency. In this case, our model successfully links the two hunks, accurately identifying their semantic association and generating a commit message that better reflects the developer's actual intent. In contrast, baseline models only capture partial information, resulting in incomplete semantic understanding in their generated commit messages.

7 Conclusion

In this paper, we propose CAMCMG, a change-aware model for commit message generation that consists of three core components. First, a local attention mechanism is designed to capture local contextual information. Second, a change-aware attention mechanism is introduced to model semantic associations across different hunks. Third, a change alignment loss is employed to explicitly align the *diff* with its commit message. In addition, we conduct a human evaluation to assess the quality of the generated commit messages, demonstrating the effectiveness of CAMCMG. Our code and data are publicly available at https://anonymous.4open.science/r/CAMCMG/.

Limitations

We have identified the following main limitations:

Generalizability to other programming languages: Our experiments are conducted on a limited set of programming languages. The generalizability of the proposed approach to other programming languages remains an open question.

Metrics: The model may generate commit messages that are semantically correct but lexically diverse, resulting in relatively low scores on the automatic metrics. Although BRSA can capture semantic similarity, the limitations of automatic metrics remain an issue for evaluation.

Change Alignment Loss: The change alignment loss relies on the presence of tokens in the *diff* that correspond to tokens in the commit message. If no such tokens exists, its effectiveness may be limited.

References

The claude 3 model family: Opus, sonnet, haiku.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, and 1 others. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Satanjeev Banerjee and Alon Lavie. 2005. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72.

Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*.

Raymond PL Buse and Westley R Weimer. 2010. Automatically documenting program changes. In *Proceedings of the 25th IEEE/ACM international conference on automated software engineering*, pages 33–42.

Maria Caulo, Bin Lin, Gabriele Bavota, Giuseppe Scanniello, and Michele Lanza. 2020. Knowledge transfer in modern code review. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 230–240.

Gonçalo M Correia, Vlad Niculae, and André FT Martins. 2019. Adaptively sparse transformers. *arXiv preprint arXiv:1909.00015*.

Michael Denkowski and Alon Lavie. 2014. Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the ninth workshop on statistical machine translation*, pages 376–380.

Jinhao Dong, Yiling Lou, Qihao Zhu, Zeyu Sun, Zhilin Li, Wenjie Zhang, and Dan Hao. 2022. Fira: fine-grained graph-based code change representation for automated commit message generation. In *Proceedings of the 44th international conference on software engineering*, pages 970–981.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407.

Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering*, pages 422–431. IEEE.

Tudor Gîrba and Stéphane Ducasse. 2006. Modeling history to analyze software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(3):207–236.

Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. 2023. Longcoder: A long-range pre-trained language model for code completion. In *International Conference on Machine Learning*, pages 12098–12107. PMLR.

Yichen He, Liran Wang, Kaiyi Wang, Yupeng Zhang, Hang Zhang, and Zhoujun Li. 2023. Come: Commit message generation with modification embedding. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 792–803.

Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, pages 518–529.

Yuan Huang, Nan Jia, Hao-Jie Zhou, Xiang-Ping Chen, Zi-Bin Zheng, and Ming-Dong Tang. 2020. Learning human-written commit messages to document code changes. *Journal of Computer Science and Technology*, 35(6):1258–1277.

Aaron Imani, Iftekhar Ahmed, and Mohammad Moshirpour. 2024. Context conquers parameters: Outperforming proprietary llm in commit message generation. *arXiv preprint arXiv:2408.02502*.

Chun-Liang Li, Kihyuk Sohn, Jinsung Yoon, and Tomas Pfister. 2021. Cutpaste: Self-supervised learning for anomaly detection and localization. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9664–9674.

Cong Li, Zhaogui Xu, Peng Di, Dongxia Wang, Zheng Li, and Qian Zheng. 2024. Understanding code changes practically with small-scale language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 216–228.

Yingling Li, Yuhan Wu, Zi'ao Wang, Lei Huang, Junjie Wang, Jianping Li, and Minying Huang. 2025. Codedoctor: multi-category code review comment generation. *Automated Software Engineering*, 32(1):25.

Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. 2023. Cct5: A code-change-oriented pre-trained model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1509–1521.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, and 1 others. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*.

Shangqing Liu, Cuiyun Gao, Sen Chen, Lun Yiu Nie, and Yang Liu. 2020. Atom: Commit message generation based on abstract syntax tree and hybrid ranking. *IEEE Transactions on Software Engineering*, 48(5):1800–1817.

Zhongxin Liu, Zhijie Tang, Xin Xia, and Xiaohu Yang. 2023. Ccrep: Learning code change representations via pre-trained code model and query back. In *2023 IEEE/ACM 45th International Conference on Software Engineering*, pages 17–29. IEEE.

Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: how far are we? In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 373–384.

Cristina V Lopes, Vanessa I Klotzman, Iris Ma, and Iftekar Ahmed. 2024. Commit messages in the age of large language models. *arXiv preprint arXiv:2401.17622*.

Walid Maalej and Hans-Jörg Happel. 2010. Can development work describe itself? In *2010 7th IEEE working conference on mining software repositories*, pages 191–200. IEEE.

Lun Yiu Nie, Cuiyun Gao, Zhicong Zhong, Wai Lam, Yang Liu, and Zenglin Xu. 2021. Coregen: Contextualized code representation learning for commit message generation. *Neurocomputing*, 459:97–107.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Hannah M. Schlüter, Jeremy Tan, Benjamin Hou, and Bernhard Kainz. 2021. Natural synthetic anomalies for self-supervised anomaly detection and localization.

Jinfeng Shen, Xiaobing Sun, Bin Li, Hui Yang, and Jiajun Hu. 2016. On automatic summarization of what and why information in source code changes. In *2016 IEEE 40th Annual Computer Software and Applications Conference*, volume 1, pages 103–112. IEEE.

Ensheng Shi, Yanlin Wang, Wei Tao, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2022. Race: Retrieval-augmented commit message generation. *arXiv preprint arXiv:2203.02700*.

Jonathan Sillito, Gail C Murphy, and Kris De Volder. 2008. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451.

Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2020. Mpnet: Masked and permuted pre-training for language understanding. *Advances in neural information processing systems*, 33:16857–16867.

Wei Tao, Yanlin Wang, Ensheng Shi, Lun Du, Shi Han, Hongyu Zhang, Dongmei Zhang, and Wenqiang Zhang. 2021. On the evaluation of commit message generation models: An experimental study. In *2021 IEEE International Conference on Software Maintenance and Evolution*, pages 126–136. IEEE.

Wei Tao, Yanlin Wang, Ensheng Shi, Lun Du, Shi Han, Hongyu Zhang, Dongmei Zhang, and Wenqiang Zhang. 2022. A large-scale empirical study of commit message generation: models, datasets and evaluation. *Empirical Software Engineering*, 27(7):198.

Wei Tao, Yucheng Zhou, Yanlin Wang, Hongyu Zhang, Haofen Wang, and Wenqiang Zhang. 2024. Kadel: Knowledge-aware denoising learning for commit message generation. *ACM Transactions on Software Engineering and Methodology*, 33(5):1–32.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Ramakrishna Vedantam, C Lawrence Zitnick, and Devi Parikh. 2015. Cider: Consensus-based image description evaluation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4566–4575.

Chuangwei Wang, Li Zhang, and Xiaofang Zhang. 2024a. Multi-grained contextual code representation learning for commit message generation. *Information and Software Technology*, 167:107393.

Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. 2021a. Context-aware retrieval-based deep commit message generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–30.

Hongqi Wang, Jiawen Zhao, Qunpo Liu, Naohiko Hanajima, and Xuhui Bu. 2025. Gfd-net: Micro-electrical connector defect localization network with global additive attention and frequency-aware feature fusion. *Signal, Image and Video Processing*, 19(11):890.

Yanlin Wang, Yanxian Huang, Daya Guo, Hongyu Zhang, and Zibin Zheng. 2024b. Sparsecoder: Identifier-aware sparse transformer for file-level code summarization. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 614–625. IEEE.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021b. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. 2025. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*.

John Wieting, Taylor Berg-Kirkpatrick, Kevin Gimpel, and Graham Neubig. 2019. Beyond bleu: training neural machine translation with semantic similarity. *arXiv preprint arXiv:1909.06694*.

Frank Wilcoxon, SK Katti, Roberta A Wilcox, and 1 others. 1963. *Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test*, volume 1. American Cyanamid Pearl River, NY.

Yifan Wu, Yunpeng Wang, Ying Li, Wei Tao, Siyu Yu, Haowen Yang, Wei Jiang, and Jianguo Li. 2025. An empirical study on commit message generation using llms via in-context learning. *arXiv preprint arXiv:2502.18904*.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and 1 others. 2020. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297.

Jian Zhang, Chong Wang, Anran Li, Wenhan Wang, Tianlin Li, and Yang Liu. 2024a. Vuladvisor: Natural language suggestion generation for software vulnerability repair. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1932–1944.

Linghao Zhang, Hongyi Zhang, Chong Wang, and Peng Liang. 2024b. Rag-enhanced commit message generation. *arXiv preprint arXiv:2406.05514*.

Yuxia Zhang, Zhiqing Qiu, Klaas-Jan Stol, Wenhui Zhu, Jiaxin Zhu, Yingchen Tian, and Hui Liu. 2024c. Automatic commit message generation: A critical review and directions for future work. *IEEE Transactions on Software Engineering*, 50(4):816–835.

A Evaluation Metrics

BLEU (Papineni et al., 2002): BLEU is an evaluation metric based on n-gram precision. It calculates a score by comparing the overlap of n-grams between the generated text and the reference text. Traditional BLEU performs poorly on short texts. Therefore, in this paper, we adopt a variant of BLEU, B-NORM (Tao et al., 2021) (the BLEU results reported in this paper are B-Norm), which introduces a normalization strategy suitable for short texts such as commit messages, allowing a more reasonable reflection of the model's actual performance.

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

where BP denotes the length penalty, which penalizes predictions that are too short. p_n represents the empirically chosen weight for each n-gram, corresponding to the matching score of the n-grams. The score is calculated as follows:

$$BP = \begin{cases} 1 & \text{if } c > 0 \\ e^{(1-r/c)} & \text{if } c \leq 0 \end{cases},$$

where c and r denote the lengths of the generated text and the reference text, respectively.

ROUGE (Lin, 2004): A metric focuses on recall rather than precision. It measures how much information in the reference text is covered by the generated text. ROUGE-L calculates the score based on the precision and recall of the Longest Common Subsequence (LCS) between the generated and reference texts. The score is calculated as follows:

$$\text{ROUGE-L} = \frac{(1+\beta^2)R_{lcs}P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}}$$

where R_{lcs} and P_{lcs} represent the recall and precision, respectively. In DUC, β is set to a very large value, so only R_{lcs} is effectively considered.

The recall and precision are defined as:

$$R_{lcs} = \frac{LCS(X,Y)}{m}, \quad P_{lcs} = \frac{LCS(X,Y)}{n}$$

where $LCS(X,Y)$ denotes the length of the longest common subsequence between X and Y, and m and n are the lengths of the reference text and the generated text, respectively.

METEOR (Banerjee and Lavie, 2005): A word-based metric used to measure the extent to which the generated text covers the reference text. It calculates a score by matching words between the generated and reference texts. Higher scores indicate greater similarity between the generated and reference texts, and hence better quality. The score is calculated as follows:

$$METEOR = (1 - Pen)F_{mean}$$

where Pen is calculated according to the number of chunks (ch) and the number of matches (m):

$$Pen = \gamma * \left(\frac{ch}{m}\right)^{\beta}$$

The values of β and γ parameters are set to 0.20 and 0.60, respectively, following the prior work (Denkowski and Lavie, 2014).

CIDEr (Vedantam et al., 2015): It measures the similarity between the generated text and reference texts by computing TF-IDF weighted n-gram similarities, and then averages the scores across different n-grams to obtain the final evaluation result. The score is calculated as follows:

$$CIDEr_n(c_i, s_i) = \frac{\langle g^n(c_i), g^n(s_i)\rangle}{\|g^n(c_i)\|\|g^n(s_i)\|},$$

where $g^n(s_i)$ is the vector formed by $g_k(s_i)$, containing all n-grams (n ranges from 1 to 4). c_i denotes the i-th generated sentence. Finally, the overall CIDEr score is computed by combining the scores from different n-grams, as follows:

$$CIDEr(c_i, s_i) = \sum_{n=1}^{N} \omega_n CIDEr_n(c_i, s_i)$$

BRSA (Li et al., 2024): This metric utilizes MPNet (Song et al., 2020), specifically *all-mpnet-base-v2* to compute the semantic similarity between the reference and generated texts, denoted as SEMSIM. To mitigate the limitations imposed by BLEU and ROUGE-L, Song et al. (Song et al., 2020) apply a double weighting on SEMSIM and combine it with ROUGE-L and BLEU using weights of 0.25, 0.25, and 0.5, respectively, to calculate the BRSA score.

$$BRSA = 0.25 \times BLEU + 0.25 \times ROUGE\text{-}L + 0.5 \times SEMSIM$$

B Dataset

Table 4 presents the statistics of commits and their corresponding hunks across five programming languages and three dataset splits (Training, Valid,

and Test). It can be observed that, on average, each commit contains approximately 3 to 5 hunks. These statistics provide a foundational basis for the subsequent analysis and modeling in this study.

Language	Phase	Commits	Hunks	Avg
Java	Training	160,018	591,633	3.70
	Valid	19,825	74,294	3.75
	Test	20,159	78,714	3.90
C#	Training	149,907	677,062	4.52
	Valid	18,688	87,777	4.70
	Test	18,702	81,009	4.33
C++	Training	160,948	739,925	4.60
	Valid	20,000	95,139	4.76
	Test	20,141	87,922	4.37
Python	Training	206,777	953,388	4.61
	Valid	25,912	128,373	4.95
	Test	25,837	117,231	4.54
JavaScript	Training	197,529	806,673	4.08
	Valid	24,899	120,697	4.85
	Test	24,773	92,523	3.73

Table 4: Statistics of the dataset.

C Questionnaire of Human Evaluation

We designed a survey for human evaluation in which each participant was asked to score the commit messages generated by each model with respect to a given code change across three dimensions: informativeness, conciseness, and expressiveness. Scores ranged from 1 to 5, with higher values indicating better quality. Figure 5 illustrates an example question from the survey, where the order of the commit messages generated by different models was randomized to avoid bias.

Figure 5: The example of questionnaire.

D Hyperparameter Experiments

In addition, we further conducted experiments on two key hyperparameters, namely the window size and the weighting factor λ, to investigate their effects on model performance. For the window size parameter, we set the values to 32, 64, 128, and 256 to analyze how different local context ranges influence the effectiveness of the sparse attention mechanism. As shown in Figure 6(a), the model achieves its best performance when the window size is set to 64. When the window size is too small, the model lacks sufficient contextual information to understand a code change. Conversely, an excessively large window introduces redundant context and increases attention noise, thereby diluting the model's focus on essential regions.

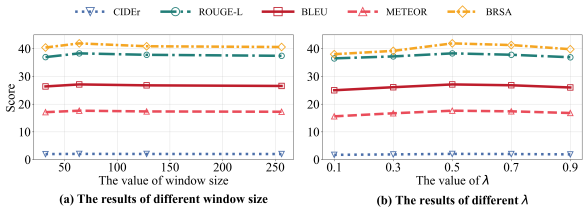

Figure 6: The results of hyperparameter experiments.

For the parameter λ, we vary its value from 0.1 to 0.9 with a step size of 0.2 to examine its role in balancing the change alignment loss and the standard cross-entropy loss (Equation 8). A smaller λ causes the model to rely more on the standard cross-entropy loss, which tends to optimize the overall linguistic quality of the generated messages, while a larger λ emphasizes the change alignment loss, thus enhancing the semantic alignment between the *diff* and its commit messages. As shown in Figure 6(b), the model performs best at $\lambda = 0.5$, suggesting that assigning a moderate weight to the change alignment loss can effectively improve the semantic consistency between code changes and commit messages.

Answering to hyperparameter experiments: We conducted hyperparameter experiments on the window size and λ, and the results show that the model achieves its best performance when $\lambda = 0.5$ and the window size is set to 64.

E Full Case

In Section 6, due to the excessive length of the diff, only a partial *diff* is shown in the case1. The complete *diff* can be found in the Figure 7.

```
@@ -521,7 +521,7 @@ public String getCode() {
     final int getCodeLength() {
-        return content() == null ? getCode().length() : content().getCodeLength();
+        return content().getCodeLength();
     }
@@ -748,7 +748,7 @@ public final LineLocation createLineLocation(int lineNumber) {
      * different source types.
      */
     Object getHashKey() {
-        return content() == null ? getName() : content().getHashKey();
+        return content().getHashKey();
     }
     final TextMap getTextMap() {
@@ -815,29 +815,23 @@ final boolean equalMime(Source other) {
         return mimeType.equals(other.mimeType);
     }

-    @Override
-    public boolean equals(Object obj) {
-        if (obj instanceof Source) {
-            Source other = (Source) obj;
-            if (content() == null) {
-                return super.equals(obj);
-            }
-            return content().equals(other.content()) && equalMime(other);
+    private static class Impl extends Source implements Cloneable {
+        Impl(Content content) {
+            super(content);
         }
-        return false;
-    }

-    @Override
-
-        if (content() == null) {
-            return super.hashCode();
+        @Override
+        public boolean equals(Object obj) {
+            if (obj instanceof Source) {
+                Source other = (Source) obj;
+                return content().equals(other.content()) && equalMime(other);
+            }
+            return false;
         }
-        return content().hashCode();
-    }

-    private static class Impl extends Source implements Cloneable {
-        Impl(Content content) {
-            super(content);
+        @Override
+        public int hashCode() {
+            return content().hashCode();
         }
     }
}
```

```
Commit message:
 Reference:    Always delegate to content() as it has to be always non-null
 NNGen:        Source . newFromTest is supported
 RACE:         merge pull request in g / truffle from ~ michael . van . de . vanter_oracle . com
 COME:         merge pull request in g / truffle from paw / fix - source - equals to master
 CodeT5-base:  Merge pull request in G / truffle from bugfix / SourceImpl to master
 CCT5:         each instances of source needs its content ( ) .
 Claude3:      Refactor Source class to delegate equals and hashCode to Content
 DeepSeekV2:   Updated getCode and getHashKey methods to use newContent method
 Llama3:       Refactor Source.java to use content().equals() and content().hashCode()
 CodeQwen:     Refactor and optimize Source class for better performance and readability.
 GPT-4:        Simplify Source methods by removing unnecessary null handling for content()
 CAMCMG:       remove redundant null checks for content() in Source.equals() and hashCode()
```

Figure 7: The complete *diff* of case1.