Leveraging Context-Aware Prompting for Commit Message Generation

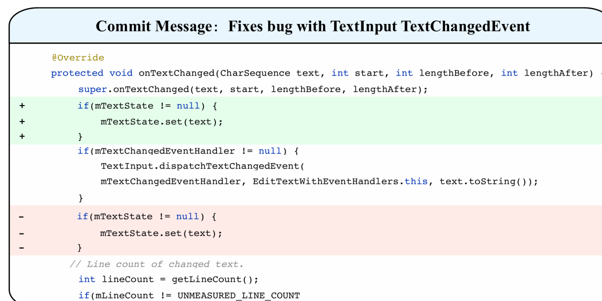
Zhihua Jiang[1], **Jianwei Chen**[1], **Dongning Rao**[2]*, **Guanghui Ye**[3]
[1] Department of Computer Science, Jinan University, Guangzhou 510632, China
[2] School of Computer, Guangdong University of Technology, Guangzhou 510006, China
[3] College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China
tjiangzhh@jnu.edu.cn, chenjw@stu2021.jnu.edu.cn, raodn@gdut.edu.cn, yghui@hnu.edu.cn

Abstract

Writing comprehensive commit messages is tedious yet important, because these messages describe changes of code, such as fixing bugs or adding new features. However, most existing methods focus on either only the changed lines or nearest context lines, without considering the effectiveness of selecting useful contexts. On the other hand, it is possible that introducing excessive contexts can lead to noise. To this end, we propose a code model COMMIT (Context-aware prOMpting based comMIt-message generaTion) in conjunction with a code dataset CODEC (COntext and metaData Enhanced Code dataset). Leveraging program slicing, CODEC consolidates code changes along with related contexts via property graph analysis. Further, utilizing CodeT5+ as the backbone model, we train COMMIT via context-aware prompt on CODEC. Experiments show that COMMIT can surpass all compared models including pre-trained language models for code (code-PLMs) such as CommitBART and large language models for code (code-LLMs) such as Code-LlaMa. Besides, we investigate several research questions (RQs), further verifying the effectiveness of our approach. We release the data and code at: https://github.com/Jnunlplab/COMMIT.git.

1 Introduction

Commit messages (Liu et al., 2022b) are communication channels of programmers and part of version control system, which tracks and manages every modification to software code during development. While version control systems help software teams work better by comparing earlier versions of code and turning back the clock to fix mistakes, natural language commit messages summarize what happened in the code change and/or explain why the code change was made (Guo et al., 2022). However, developers, whose time is precious, are often reluctant to write commit messages.


Commit Message: Fixes bug with TextInput TextChangedEvent

```
    @Override
    protected void onTextChanged(CharSequence text, int start, int lengthBefore, int lengthAfter) {
        super.onTextChanged(text, start, lengthBefore, lengthAfter);
+       if(mTextState != null) {
+           mTextState.set(text);
+       }
        if(mTextChangedEventHandler != null) {
            TextInput.dispatchTextChangedEvent(
            mTextChangedEventHandler, EditTextWithEventHandlers.this, text.toString());
        }
-       if(mTextState != null) {
-           mTextState.set(text);
-       }
        // Line count of changed text.
        int lineCount = getLineCount();
        if(mLineCount != UNMEASURED_LINE_COUNT
```

Figure 1: Motivation example of this paper.

Five kinds of automatic commit message generation approaches have been proposed. First, information retrieval based models (Liu et al., 2018) find the most similar messages for a new commit. Second, end-to-end generative models (Liu et al., 2019) are trained directly on task corpus. Third, hybrid models (Wang et al., 2021a) incorporate both retrieved and generated messages. Fourth, pre-trained language models (PLMs) are adapted into code tasks (Liu et al., 2022b). Fifth, large language models (LLMs) are called via APIs or fine-tuned (OpenAI, 2023). However, existing methods normally have some shortcomings, e.g., focusing on only the changed lines is not enough or including unrelated lines around changes will bring in noise (Dong et al., 2022a; Liu et al., 2022b). As exemplified in Fig. 1, the added and deleted codes are literally the same, leading to the code changes ill-defined. Also, the association between the changed lines and the unchanged parts of the program is unclear due to lacking program dependency analysis.

Motivated by the fact that many static analysis tools[1] have been proposed based on the theory of code property graphs (CPGs) (Yamaguchi et al., 2014), we propose a code model COMMIT (Context-aware prOMpting based comMIt-message generaTion) in conjunction with a code dataset CODEC (COntext and metaData Enhanced

*Corresponding author: Dongning Rao.

[1] https://github.com/wimkeir/graft

Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pages 13522–13540
November 12-16, 2024 ©2024 Association for Computational Linguistics

Code dataset). We construct CODEC for consolidating the code changes along with their semantically related contexts via using CPGs. Further, designing context-enhanced prompt, we implement COMMIT using CodeT5+ (Wang et al., 2023) as the backbone model that is trained on CODEC.

The technique novelty is two-fold. First, we propose new graph representations including Deleted Context Graphs (DCGs), Added Context Graph (ACGs), and Added-Deleted Context Graph (ADCGs). Both DCGs and ACGs can be efficiently derived from CPGs and we can further construct ADCGs via combining ACGs and DCGs (e.g., identifying the same program statements and collecting the dependency edges). ADCGs contain richer information than both DCGs and ACGs. Based on them, we can efficiently extract useful contexts (indicated by ADCG nodes) from code differences (i.e., Diffs) via program slicing. Second, we select possibly crucial contexts which have a CPG-contained program dependency with the changed codes, and enhance a prompt-based code model (e.g., CodeT5+) for better message generation.

Our contributions can be summarized as follows:

- We construct a context-enhanced code dataset CODEC for commit message generation.
- We propose a context-aware code model COMMIT. Specifically, we introduce the new graph representations ADCGs based on CPGs, propose a dependency extraction algorithm, design the context-aware prompt, and implement the proposed COMMIT using CodeT5+ as backbone that is trained on CODEC.
- Experimental results show that our model COMMIT can outperform 16 compared models including code-PLMs/LLMs, and reach the highest scores regarding multi reference-based metrics, e.g., 13.13 on BLEU-4, 12.48 on METEOR and 24.05 on ROUGE-L.

The rest of this paper is organized as follows. First, after introducing CPGs in Sec. 2, the collection process of CODEC, along with a comparison with seven commonly used datasets, are placed at Sec. 3. Then, COMMIT is presented in Sec. 4. The experiments in Sec. 5 include comparisons between various models and discusses on RQs as well as case studies. We conclude this paper with future works in (Sec. 6). More details including the experimental settings and the prompt design are posted in Appx. A∼D because of the space limit.

2 Code Property Graph

CPGs play a vital role in extracting associated context of the code change in our method. Following definitions from previous studies (Yamaguchi et al., 2014), CPGs can be formally defined as follows.

Definition 1 *Property Graph (PG).* The property graph $G = (V, E, \lambda, \mu)$ is a labeled directed pseudograph with attached properties. V is the set of all vertices, $E \subseteq (V \times V)$ is the edge set, and $\lambda : E \to \Sigma$ is the edge label function which assigns a label from the Alphabet Σ to an edge. Given all keys of properties K and all values of properties S, $\mu : (V \cup E) \times K \to S$ is the property assign function which assigns properties to either a vertex or an edge. E.g., key-value pairs can be (k, x).

Definition 2 *Abstract Syntax Tree (AST).* The property graph of *AST* is $G_A = (V_A, E_A, \lambda_A, \mu_A)$, where V_A is the tree nodes set and E_A is the tree edges set. While keys of G_A are in $K_A = \{code, order\}$, and the value $x \in \mathbb{N}$.

Definition 3 *Control Flow Graph (CFG).* The property graph of *CFG* is $G_C = (V_C, E_C, \lambda_C, \mu_C)$. V_C is the set of all lines of code whose type in *AST* is a statement or expression. While λ_C is the edge label function and $\Sigma_C = \{true, false, \epsilon\}$.

Definition 4 *Program Dependence Graph (PDG).* The property graph of *PDG* is $G_P = (V_P, E_P, \lambda_P, \mu_P)$, where $\lambda_P : E_P \to \Sigma_P$ and $\Sigma_P = \{C, D\}$ and C and D stand for control flow data flow respectively. Additional property symbols can be added to dependencies as a predicate-state-indicator (i.e., true/false).

Combining the above three graphs, we have *CPG* defined as $G = (V, E, \lambda, \mu)$, where $V = V_A$, $E = E_A \cup E_C \cup E_P$, $\lambda = \lambda_A \cup \lambda_C \cup \lambda_P$, and $\mu = \mu_A \cup \mu_C \cup \mu_P$.

3 The CODEC Dataset

Since existing datasets lack supervision signals on context of the code change, we construct our own dataset for the studied task in this paper. We elaborate on main steps of dataset construction and provide a comprehensive comparison with others.

3.1 Dataset Construction

Collection. With PyDriller[2], we collect Java repositories with at least 1000 stars on GitHub. Alongside source code changes, the context of the changes is also recorded. E.g., related files, the status of the code before and after the change.

[2]https://pydriller.readthedocs.io

	CmtGen	NNGen	CoDiSum	PtrGNCMsg	MultiLang	ATOM	MCMD	CODEC
# Train Set [1]	26,208	22,112	75,000	23,623	122,756	160,354	1,800,000	10,301
# Valid Set	3,000	2,511	8,000	5,051	15,344	19,796	225,000	3,433
# Test Set	3,000	2,521	7,661	3,989	15,344	17,818	225,000	3,435
# Repositories	1,000	1,000	1,000	2,000	12	56	500	160
Reproducible	×	×	×	×	✓	×	✓	✓
Deduplicated	×	✓	×	×	✓	✓	✓	✓
Filtering	✓	✓	×	×	×	✓	✓	✓
Context[2]	×	×	×	×	×	×	×	✓
Content	Diff Message	Diff Message	Diff Message	Diff Message	Diff Message Repo	Diff Message RepoName SHA Timestamp	Diff Message RepoFullName SHA Timestamp	Diff Message Repo SHA **Context**

[1] # : number of items.
[2] Additional information of context and metadata includes Owner, ChangeLines, AddLines, DeleteLines, NegativeChanges, PositiveChanges, FileName, BeforeCode and AfterCode.

Table 1: Dataset Comparisons.

Filtering. Four filtering techniques are applied in CODEC. (1) Uncontrollable changes. When the number of changed lines is big, most automatically generated comments are unreliable. I.e., changes over 20 lines are ignored. (2) Length limitation. Commit message should have upper and lower length bound. E.g., a message less than five words is uninformative, while a message greater than 150 words might be confusing. (3) Only Java file changes. (4) Removing unrelated information. E.g., issueID, commitID, and URL.

Optimization. Following previous studies (Gu et al., 2016; Jiang et al., 2017), only the first sentence of commit message is considered. We also build a Bi-LSTM (Cornegruta et al., 2016) model to keep only information that focuses on Why/What, as recent studies suggested (Tian et al., 2022).

Statistics of CODEC is presented in Tab. 11, Appx. A. We also provide an example of commit message and its data items collected by PyDriller in Fig. 5 and Fig. 6, Appx. A, respectively.

3.2 Dataset Comparisons

Tab. 1 compares CODEC with other seven datasets.

CmtGen (Jiang et al., 2017). It is a small dataset based on the top 1000 Java repositories. CmtGen introduces Verb-Direct Object based filtering, but it still contains robot-generated information and duplicate data.

NNGen (Liu et al., 2018). NNGen is the cleaned-up version of CmtGen. It removes trivial and robot-generated commit messages.

CoDiSum (Xu et al., 2019). With a file-level programming language filter, CoDiSum is also based on the top 1000 Java repositories.

PtrGNCMsg (See et al., 2017). PtrGNCMsg uses the same pre-processing as CmtGen, but contains the top 2000 Java repositories instead of 1000.

MultiLang (Loyola et al., 2017). While it contains three programming languages from 12 hand-selected repositories, it is not open-source.

ATOM (Liu et al., 2020). Gathering data from the 56 Java projects with the most stars, ATOM filters out commits with noisy information and commits that do not contain source code changes. This dataset provides not only the original commits but also the affected functions in each commit.

MCMD (Tao et al., 2021). With no filtering techniques, this large dataset contains programs in five programming languages from the top 500 repositories on GitHub.

From the comparisons, we find that even though CODEC is relatively small-scale, it provides context-enhanced training corpus for our approach.

4 The COMMIT Model

Implementing COMMIT has four steps: (1) CPGs are built by analyzing source codes before and after changes. (2) Dependencies in CPG are first extracted and then merged to ADCG. (3) Context-aware representation is constructed on ADCG via programming slicing. (4) Commit-message is generated by context-aware prompting on CodeT5+.

4.1 Overview of Our Model

Fig. 2 provides the overview of COMMIT. After data collection (from GitHub and with pre-processes, see the top part of Fig. 2), we build CPGs and ADCG to acquire context enhanced representation. Then COMMIT learns a continuous

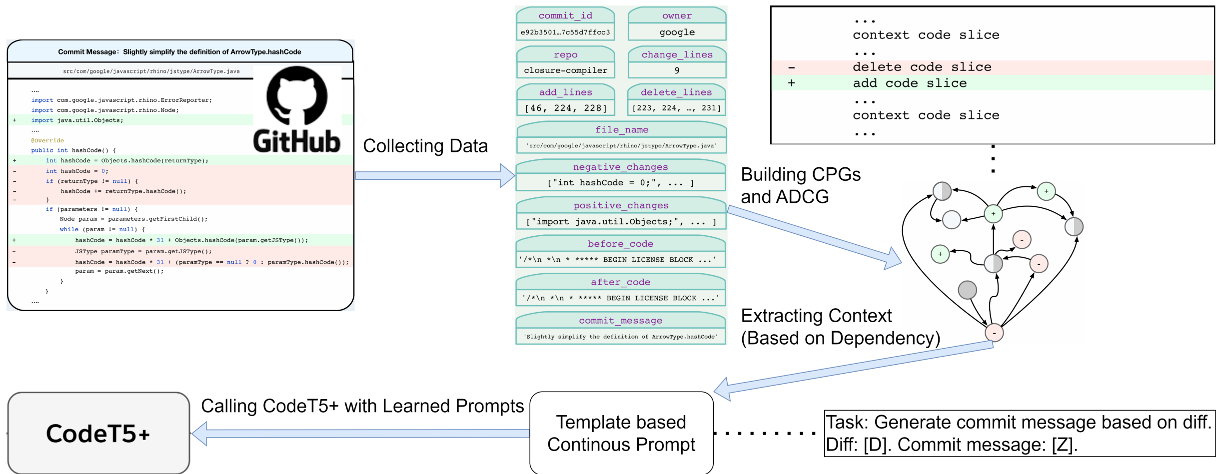


Figure 2: Overview of COMMIT.

prompt (starting from manually designed templates and enhancing with extracted context, see the bottom part of Fig. 2). At last, CodeT5+ is responsible for the generation of commit messages.

4.2 Building Added-Deleted Context Graph

We use a static code analysis tool, Joern[3], to build two CPGs for the code before and after changes. Particularly, these two CPGs are called CPG_{delete} for deleted lines and CPG_{add} for added lines, respectively. In CPG_{delete}, we annotate nodes, which are corresponding to deleted lines, with a "-" symbol (red vertices in Fig. 3). In CPG_{add}, we annotate nodes, which are corresponding to added lines, with a "+" symbol (green vertices in Fig. 3).

Inspired by code differences (Diffs for short) which are generated via combining the codes before and after the changes, we propose ADCGs which are generated via combining ACGs and DCGs, facilitating the extraction of relevant contexts from Diffs. We identify the statement lines with the same content and collect all dependency edges from DCGs and ACGs. Thus, ADCGs contain richer information. There are five kinds of nodes (see Fig. 3): 1) deleted lines (e.g., line 8 in red); 2) added lines (e.g., line 8 in green); 3) context lines relevant with deletion (e.g., line 2 in dark gray); 4) context lines relevant with addition (e.g., line 3 in light gray); 5) context lines relevant with both (e.g., line 11, half dark and half light). Using ADCGs, we can extract useful context statements indicated by ADCG nodes from Diffs via program slicing. We elaborate on the two stages below.

Identifying Dependency. Considering both *data dependency* and *control dependency*, COMMIT extracts these two kinds of dependencies from CPG_{add} and CPG_{delete} according to Alg. 1. On contrast, data dependency is a relationship in which a program instruction refers to the data of a preceding statement, while control dependence affects the flow of control in a program from instruction to instruction through the existence of branches. The extraction results are stored in new graphs: deleted context graph (DCG) and added context graph (ACG), only dependency-related nodes kept.

Merging vertices and edges. After first merging the identical vertices in ACG and DCG, COMMIT then integrates all dependency edges within specified depth. Then, we have the $ADCG$ (the right part of Fig. 3).

4.3 Dependency Extraction Algorithm

Considering the negative effects of over-smoothing and balancing between model's complexity and efficiency, we limit the scope of dependencies to a certain depth. I.e., Alg. 1 identifies context statements with either data dependency or control dependency, which are reachable to the changed statement from $ADCG$ within a specific depth.

After initialization in lines 1∼4, changed lines are in $nodes_d$, related statements along with the connections between these nodes are in $edges_d$, and program slicing are made. Then lines 5∼8 build the source and target mappings. The *slicing* is then iteratively visited to accomplish the merge, between lines 9∼23. While all related line numbers should be added into $result$, the changed lines need be removed from $result$ (lines 24∼28).

[3]https://docs.joern.io/code-property-graph

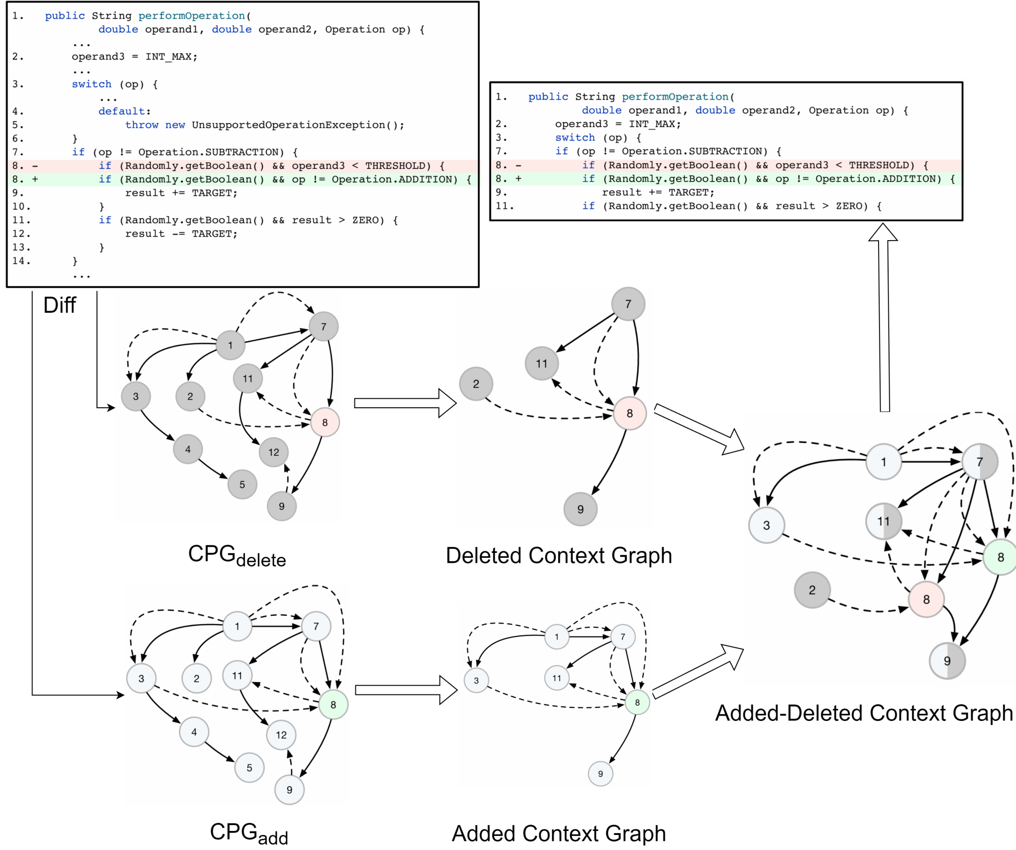


Figure 3: An example of extracting context from $ADCG$ via Alg. 1, where a solid line indicates data dependency and a dotted line indicates control dependency. With program slicing, the selected context statements (e.g., lines 2, 3 and 7) as well as the changed statements (e.g., lines +8 and -8) from source code are kept as input in our method.

Algorithm 1 Dependency Extraction Algorithm

Require: Changed lines $change_nodes$, ADCG vertices $nodes$, ADCG edges $edges$, Depth of dependency $dependence_depth$
Ensure: Line numbers of dependencies $result$

1: $map_{src}, map_{tgt}, result \leftarrow \emptyset$
2: $nodes_d \leftarrow change_lines$
3: $edges_d \leftarrow$ statements of data or control dependency
4: $\forall src, tgt \in nodes_d$, make $slicing(src, tgt, edges, 0)$
5: **for** all $e(v, u) \in edges$ **do**
6: $\quad map_{src} \leftarrow map_{src} \cup e'(v, u'), u' \in nodes$
7: $\quad map_{tgt} \leftarrow map_{tgt} \cup e'(v', u), v' \in nodes$
8: **end for**
9: **while** $slicing \neq \emptyset$ **do**
10: $\quad ele \leftarrow$ the first element of $slicing$
11: $\quad$**if** $element.depth \leq dependence_depth$ **then**
12: $\quad\quad$**continue**
13: $\quad$**end if**
14: $\quad$**if** $ele.src($ or $ele.tgt) \neq \emptyset$ **then**
15: $\quad\quad$**for** all $e(v', u') \in map_{src}(map_{tgt})$ **do**
16: $\quad\quad\quad$**if** v' or $u' \notin nodes_d$ **then**
17: $\quad\quad\quad\quad nodes_d \leftarrow nodes_d \cup v'$ or u'
18: $\quad\quad\quad\quad i \leftarrow i + 1$
19: $\quad\quad\quad\quad$add $slicing(v', u', edges_d, i)$
20: $\quad\quad\quad$**end if**
21: $\quad\quad$**end for**
22: $\quad$**end if**
23: **end while**
24: **for** all $v \in nodes_d$ **do**
25: $\quad result \leftarrow$ the line number of v
26: **end for**
27: $result \leftarrow result - change_lines$
28: **return** $result$

4.4 Prompt Design

Prompts for commit message generation in previous studies (Liu et al., 2022b; Wang et al., 2023) are simple, e.g., "Diff: [D], Commit message: [Z]". We extend the common prompts into four specific templates $T1 \sim T4$, as shown in Tab. 2. Each template is composed of task prompt, code differences and ground-truth commit message. All information is concatenated in a textual prompt template.

No	Template
T1	Task: Generate commit message based on diff. Diff: [D]. Commit message: [Z].
T2	Commit message generation: Given diff, generate its commit message. Diff: [D]. Commit message: [Z].
T3	Generate commit message for diff: [D]. Commit message: [Z].
T4	Given diff: [D]. Generate its commit message: [Z].

Table 2: Prompt Design for CodeT5+.

For input x, let the prompt word of features $a_i \in A$ of x be d_i and the prompt function is in Eq. 1.

$$f_{\text{prompt}}(a_i) = d_i : [a_i] \quad 1 \leq i \leq |A| \qquad (1)$$

For the code change (i.e., diff) and commit message, the prompts are in Eq. 2 and Eq. 3.

$$f_{\text{prompt}}(a_d) = d_d : [a_d] = \text{Diff: [D]} \quad (2)$$

$$\begin{aligned} f_{\text{prompt}}(a_{cm}) &= d_{cm} : [a_{cm}] \\ &= \text{Commit message: [Z]} \end{aligned} \quad (3)$$

Let $f_{fill}(a_t)$ be the task prompt, where a_t is a task description, the full prompt for x is $f_{\text{prompt}}(x) = f_{fill}(a_t).f_{\text{prompt}}(a_d).f_{\text{prompt}}(a_{cm})$. Then, our optimization target is defined as Eq. 4, where S is the generated message.

$$P(\text{S}|\text{X}, f_{\text{prompt}}) = \sum_{x \in \text{X}, z \in [\text{Z}]} \left(\max \prod_{j=1}^{|z|} P_\theta(z_j | x, f_{\text{prompt}}, z_{<j}) \right) \quad (4)$$

5 Experiments

5.1 Basic Settings

Experiment settings can be located in Tab. 12, Appx. B. We ran all experiments on two Nvidia GeForce RTX™ 3090 GPUs. We set the hyperparameter "depth" to 3 be default after some probing tests (see Tab. 5 for more details).

5.2 Compared Models

COMMIT is compared with 16 models in this paper. These models can be recognized as five types.

(1) *Information retrieval* based models:

NNGen (Liu et al., 2018). NNGen combines the nearest neighbor algorithm and the "bag of words" model to retrieve instances that are most similar to the target code (e.g.,facilitating cosine similarity).

Lucene[4]. It is a popular search engine.

(2) *End-to-end* generative models:

PtrGNCMsg (Liu et al., 2019). PtrGNCMsg leverages graph neural networks.

CoreGen (Nie et al., 2021). CoreGen is a two-stage context-aware method. It first learns contextualized code representations from commit sequences and then fine-tunes with downstream tasks. Both stages are implemented based on Transformer.

FIRA (Dong et al., 2022b). FIRA represents the code changes via leveraging fine-grained program graph structures. It includes a GNN-based encoder and a transformer-based decoder.

(3) *Hybrid* models:

CoRec (Wang et al., 2021a).Considering that generative models tend to generate high-frequency words and ignore low-frequency words, as well as the exposure bias problem, CoRec aggregates both generative models and retrieval models.

RACE (Ahmad et al., 2021). RACE generates commit messages based on retrieval enhancement. It retrieves similar messages as examples to guide the model and learns the intention of changes.

ATOM (Liu et al., 2022a). ATOM incorporates AST for representing diffs and integrates both retrieved and generated messages via hybrid ranking. It encodes AST paths from diffs with BiLSTM and implements the ranking module via CNN.

(4) Code-PLMs:

CodeBERT (Feng et al., 2020). CodeBERT is a variant of BERT adapted for code tasks.

CommitBERT (Jung, 2021). Based on CodeBERT, CommitBERT is fine-tuned for the task of commit message generation.

CommitBART (Liu et al., 2022b). Using denoising, cross modal generation and contrastive learning, CommitBART is based on PLBART (Ahmad et al., 2021) and designed for GitHub.

CodeT5 (Wang et al., 2021b). CodeT5 is a variant of T5 (Raffel et al., 2020), which focuses on the bi-directional relations between code and text.

UnixCoder (Guo et al., 2022). This is a cross-modal PLM that maps abstract syntactic trees and comments to sequential representations.

CodeT5+ (Wang et al., 2023). CodeT5+ is the cross-modal and enhanced version of CodeT5.

(5) LLMs or code-LLMs:

GPT-3.5-Turbo (OpenAI, 2023). A powerful LLM with parameter size of 175B.

Code-LlaMa (Rozière et al., 2023). Code-LlaMa is adapted from LlaMa-7B (Touvron et al., 2023) particularly for code tasks.

We design three prompts for LLMs (more information can be found in Tab. 13, Appx. C). Prompt-1 is a comprehensive and instructive prompt, Prompt-2 is a prompt that aims to generate concise and well-formatted commit messages. Prompt-3 adds a role in setting, requiring LLMs to focus on creating clear, informative, and concise commit messages. The prompts for zero-shot, one-shot, and few-shot scenarios are placed at Tab. 14, Appx. C. It should be noted that we randomly selected few-shot examples from the training set. For GPT-3.5 and Code-LlaMa, we report their best results in our experiments for primary comparisons. A full report of LLMs results can be found in Tab. 8∼9.

[4]https://lucene.apache.org/

5.3 Metrics

Common evaluation metrics used for generation tasks include BLEU (Papineni et al., 2002), METEOR (Banerjee and Lavie, 2005) and ROUGE (Lin, 2004). Following previous works, we use BLEU-4 (B-4), METEOR (M) and ROUGE-L (R-L). BLEU-4 scores by matching quadruple pairs, taking the coherence and fluency at the phrase level into consideration. It is more suitable for evaluating the quality of summaries. ROUGE calculates the overlap between two texts, including characters, words and sentences. By using the Longest Common Subsequence, ROUGE-L can better capture long-distance dependencies. METEOR improves BLEU by better aligning with human judgments.

5.4 Results and Analysis

In this section, we investigate seven RQs to validate the effectiveness and superiority of our approach.

RQ1: Comparisons with other methods

Model	BLEU-4	METEOR	ROUGE-L
NNGen	10.63	8.3	16.42
Lucene	9.81	7.99	15.36
PtrGNCMsg	10.39	6.79	20.00
CoreGen	10.58	8.72	18.70
FIRA	11.71	10.24	19.86
CoRec	10.85	8.26	16.86
RACE	11.43	8.74	20.41
ATOM	10.95	8.90	18.08
CommitBERT	10.03	7.02	16.94
CommitBART	11.55	8.58	18.32
CodeBERT	10.20	7.15	17.44
CodeT5	12.33	10.58	22.58
UnixCoder	11.13	8.15	18.23
CodeT5+	12.92	11.56	23.35
GPT-3.5-Turbo	11.25	10.28	21.22
Code-LlaMa-7B	2.38	7.5	9.42
COMMIT (ours)	**13.13**	**12.48**	**24.05**

Table 3: Model Comparison.

Our model achieves the best performance regarding three evaluation metrics, as shown in Tab. 3. It reaches 13.13 on BLEU-4, 12.48 on METEOR, and 24.05 on ROUGE-L. In comparison, the performance of other methods is relatively low. Information retrieval based models are fast but lack a deep understanding of code changes, so they have low scores on BLEU-4 and METEOR. End-to-end models are insufficient in capturing long-distance dependencies and complex language structures, so they have a lower METEOR. Hybrid models combine the advantages, but their ROUGE-L scores still need improvements. PLMs/LLMs based models surpasses the others except for ours.

RQ2: Effectiveness of CPG

To explore the effectiveness of various dependency representation techniques, we conduct experiments using control flow graph (CFG), program dependency graph (PDG) and code property graph (CPG). Results are presented in Tab. 4.

	BLEU-4	METEOR	ROUGE-L
CFG	12.40	11.77	23.17
PDG	12.45	11.79	23.03
CPG	**13.13**	**12.48**	**24.05**

Table 4: Effectiveness of Code Property Graphs.

As can be seen from the experimental results, the CPG score on the BLEU-4 indicator is 13.13, higher than the other two. This result shows that the CPG representation has significant improvements on word order and the grammatical accuracy. METEOR, as an indicator of semantic accuracy, also has obvious advantages through our approach with CPG. The ROUGE-L score focuses on the long sequence similarity between the generated text and the reference text. The CPG representation consistently achieves a highest score 24.05 on this indicator.

RQ3: Effectiveness of Depth Setting

To analyze the impact of dependency depth on context-aware enhanced representation, we test dependency depths in a scale of {1,2,3,4,5}. See Tab. 5. The experimental results reveal a nonlinear performance trend. When the model's understanding of dependency relationship is gradually enhanced, the performance of the model improves, which is reflected in the accuracy, clarity and relevance of the commit message to the code changes.

Depth	BLEU-4	METEOR	ROUGE-L
1	12.37	12.02	23.22
2	12.68	12.32	23.34
3	**13.13**	**12.48**	**24.05**
4	12.71	12.30	23.39
5	12.57	12.32	23.16

Table 5: Hyperparameter Test.

This improvement peaks when the dependency depth reaches 3. However, further increase in dependency depth does not continue to promote performance improvement, since a significant performance drop is observed. After the dependency

depth exceeds 3, continuous increase leads to negative impact on the generation of commit information. The reason may be that with the increase of dependency depth, the computational complexity increases synchronously. When dealing with deep dependencies, information may gradually lose focus, especially when dealing with structured data such as natural language or code, those distant relationships are often not closely related to the current context, and the model is difficult to identify and extract signals that are beneficial to the current task. Finally, when the model tries to grasp the broader context, it may introduce some irrelevant or even misplaced information, which will interfere with the model's performance.

RQ4: Effectiveness of Prompt Learning

To validate the design of prompts, we conduct experiments using prompts T1∼T4 (Sec. 4.4) respectively. Results in Tab. 6 show that T1 is best-performing. However, we observe that the influence of minor variations in prompt templates was marginal. We believe that these results indicate an inherent robustness of the model to prompt variations.

Template	BLEU-4	METEOR	ROUGE-L
T1	**13.13**	**12.48**	**24.05**
T2	12.80	12.32	23.76
T3	12.83	12.32	23.74
T4	12.91	12.27	23.72

Table 6: Effectiveness of Prompt Learning.

RQ5: Effectiveness of Changed Lines

The number of changed lines indicates the complexity of problems. To test this, we divide CODEC into four subsets regarding the quantity range of changed lines. We run COMMIT on each subset and results are presented in Tab. 7. With the amount growth of changed lines, the model's performance continuously degrades.

# Changed Lines	BLEU-4	METEOR	ROUGE-L
From 1 to 5	**13.73**	**13.67**	**25.28**
From 6 to 10	12.77	12.44	24.62
From 11 to 15	12.61	12.08	23.80
From 16 to 20	11.99	11.30	21.76

Table 7: Effect of the Number Of Changed Lines.

RQ6: Effectiveness of LLMs

The design of prompts for LLMs is shown in Tab. 13. We also present the design of prompts for in-context learning (i.e., zero-shot, one-shot, and few-shot cases) in Tab. 14. All results of two LLMs GPT-3.5-Turbo and LlaMa-7B in our experiments are given in Tab. 8 and 9 respectively.

Prompts	B-4	M	R-L
Zero-shot Prompt-1	10.59	9.44	20.71
Zero-shot Prompt-2	9.24	**10.28**	20.00
Zero-shot Prompt-3	10.47	10.18	21.14
One-shot Prompt-1	**11.25**	9.53	**21.22**
One-shot Prompt-2	11.11	9.92	21.22
One-shot Prompt-3	11.20	9.69	21.16
Few-shot Prompt-1	10.44	9.81	20.52
Few-shot Prompt-2	10.19	10.10	20.42
Few-shot Prompt-3	10.47	9.85	20.6

Table 8: Results of GPT-3.5-turbo.

Prompts	B-4	M	R-L
Zero-shot Prompt-1	**2.38**	**7.5**	**9.42**
Zero-shot Prompt-2	1.14	6.55	7.0
Zero-shot Prompt-3	1.31	6.76	6.36
One-shot Prompt-1	1.65	5.77	7.21
One-shot Prompt-2	1.37	5.22	6.76
One-shot Prompt-3	1.14	4.39	5.67
Few-shot Prompt-1	2.29	6.2	8.61
Few-shot Prompt-2	1.71	5.6	5.6
Few-shot Prompt-3	2.12	6.57	8.26

Table 9: Results of Code Llama-7B.

To conclude, GPT-3.5 in all paradigms has demonstrated the ability of generating accurate commit messages. Although different learning paradigms have varied instructions, they can understand the context to a certain extent and identify the key points of code changes. However, the commit messages generated by Code-Llama are too detailed. It not only captures the core content of code changes but also detailedly describes the code and components directly related to these changes, followed by a brief description of the purpose of code changes. Although these excessive details increase the information richness, it leads to unsatisfactory performance regarding three evaluation indicators used in this paper.

RQ7: Effectiveness of Context-rich Data

To validate the effectiveness of context-enhanced data solely, we propose three cases: *ChangeLine*, where we only use the changed lines as input; *Diff*, where we utilize the changed lines along with the Diff blocks around as input; *Context*, where we adopt the context-enhanced changed lines via Alg. 1 as input. We conduct this part of experiments using four diverse PLMs including CodeBERT, CodeT5, UnixCoder and CodeT5+, without fine-tuning. We can observe that, all tested models

obtain the maximal improvement in the *Context* case regarding all metrics.

Model	Represent	B-4	M	R-L
CodeBERT	*ChangeLine*	9.63	6.94	16.53
	Diff	10.20	7.15	17.44
	Context	**10.47**	**7.52**	**17.96**
CodeT5	*ChangeLine*	11.47	10.08	21.7
	Diff	12.33	10.58	22.58
	Context	**12.47**	**10.69**	**22.89**
UnixCoder	*ChangeLine*	10.21	7.46	18.07
	Diff	11.13	8.15	18.23
	Context	**11.17**	**8.37**	**19.52**
CodeT5+	*ChangeLine*	11.63	10.68	21.84
	Diff	12.92	11.56	23.35
	Context	**13.11**	**11.73**	**23.65**

Table 10: Effectiveness of Context Representation.

5.5 Human Evaluations

Considering the practical application of commit messages, we add human evaluations such as user studies to complement the aforementioned automatic quantitative metrics. Following recent works (Zhang et al., 2024), we first randomly select 300 samples from the test set. Then, we ask two coders to independently review each Diff and select only one commit message, which fits the code difference best, out of the 7 candidates (i.e., a representative model per category in Table 2). We adopt zero-shot prompting for two LLMs. Last, we count the number of times (i.e., the average value of two participants) selected as the best commit message and show the distribution of participants' preferences. As shown in Tab. 4, we find that in 33.2% (i.e., the highest ratio) of all the 300 samples, the commit messages generated by our model COMMIT are considered the best, further validating the advantages of our approach.

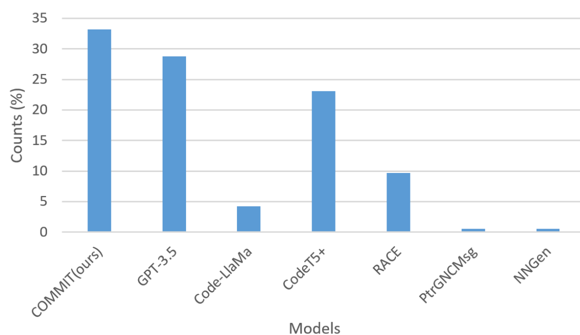


Figure 4: Human evaluations on generated messages.

5.6 Case Study on Generated Messages

We compare the commit messages generated by all compared models in this paper, given the same input instance. Results are shown in Fig. 7∼11, Appx. D, due to the writing space limit. We observe that: (1) The commit message generated by COMMIT can be completely consistent with reference, while other compared methods have their own shortcomings. See Fig. 7 for details. (2) Unexpectedly, both CodeBERT and CodeT5+ output some erroneous characters (e.g., "[jOOQ]") in generated messages. (3) GPT-3.5-Turbo demonstrates accurate commit messages, capturing the main idea of code changes (e.g., removing methods, see Fig. 8). (4) The commit messages generated by CodeLlama are lengthy and over-described (Fig. 9∼11).

6 Conclusion

This paper proposes the CODEC dataset and the COMMIT model. As dependency-sensitive contexts to the code changes are fully considered, CODEC is featured as context-aware comparing with commonly used datasets. We also propose a dependency extraction algorithm based on the innovative added-deleted context graphs and implement COMMIT using CodeT5+ as backbone. Extensive experiments and case studies demonstrate the effectiveness and superiority of our proposals.

Future works are as follows. Our current dataset collected only Java programs. We would like to extend the dataset by including other programming language codes such as C++ and Python in future. We will further validate the effective of our method on diverse programming-style codes. Besides, we will conduct an in-depth analysis on the contribution of each kind of program dependence used in this paper, to the overall performance of our model.

Limitations

We summarize the limitations of this paper here. First, our CODEC dataset only collected Java programs. By contrast, the Multilang dataset included C++ and Python codes and the MCMD dataset further took C# and JavaScript into consideration. The effectiveness of our proposal has not fully explored with other programming languages yet. Second, we identified the crucial contexts to the code changes via considering data- or control-dependence based on program property analysis. However, the unique contribution of each dependence to the model's performance has not deeply investigated.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 2655–2668.

Satanjeev Banerjee and Alon Lavie. 2005. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *IEEvaluation@ACL*.

Savelie Cornegruta, Robert Bakewell, Samuel Withey, and Giovanni Montana. 2016. Modelling radiological language with bidirectional long short-term memory networks. In *Proceedings of the Seventh International Workshop on Health Text Mining and Information Analysis*. Association for Computational Linguistics.

Jinhao Dong, Yiling Lou, Qihao Zhu, Zeyu Sun, Zhilin Li, Wenjie Zhang, and Dan Hao. 2022a. 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.

Jinhao Dong, Yiling Lou, Qihao Zhu, Zeyu Sun, Zhilin Li, Wenjie Zhang, and Dan Hao. 2022b. FIRA: fine-grained graph-based code change representation for automated commit message generation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 970–981.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, Findings of ACL, pages 1536–1547.

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 631–642.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. 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*, pages 7212–7225.

Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 135–146. IEEE.

Tae-Hwan Jung. 2021. Commitbert: Commit message generation using pre-trained programming language model. *CoRR*, abs/2105.14242.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Annual Meeting of the Association for Computational Linguistics*.

Qin Liu, Zihe Liu, Hongming Zhu, Hongfei Fan, Bowen Du, and Yu Qian. 2019. Generating commit messages from diffs using pointer-generator network. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 299–309. IEEE.

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.

Shangqing Liu, Cuiyun Gao, Sen Chen, Lun Yiu Nie, and Yang Liu. 2022a. ATOM: commit message generation based on abstract syntax tree and hybrid ranking. *IEEE Trans. Software Eng.*, 48(5):1800–1817.

Shangqing Liu, Yanzhou Li, and Yang Liu. 2022b. Commitbart: A large pre-trained model for github commits. *CoRR*, abs/2208.08100.

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.

Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A neural architecture for generating natural language descriptions from source code changes. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 2: Short Papers*, pages 287–292.

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.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*, pages 311–318. ACL.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 1073–1083.

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 (ICSME)*, pages 126–136. IEEE.

Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. 2022. What makes a good commit message? In *Proceedings of the 44th International Conference on Software Engineering*, pages 2389–2401.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288.

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.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021b. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. 2019. Commit message generation for source code changes. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pages 3975–3981.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE symposium on security and privacy*, pages 590–604. IEEE.

Linghao Zhang, Jingshu Zhao, Chong Wang, and Peng Liang. 2024. Using large language models for commit message generation: A preliminary study. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, Rovaniemi, Finland, March 12-15, 2024*, pages 126–130.

A Dataset Statistics

Statistics of CODEC is in Tab. 11. The average number of tokens in Diff messages and commit messages are 366.6 and 16.5, correspondingly.

Repo	# commit	# changed line	Avg. changes
graal	915	6,416	7.01
bazel	806	5,697	7.07
tomcat	657	4,661	7.09
buck	621	4,615	7.43
trino	589	4,206	7.14
jenkins	492	3,204	6.51
selenium	445	3,189	7.17
jmeter	428	3,166	7.4
dbeaver	416	2,809	6.75
closure-compiler	414	3,042	7.35
150 projects more			
total	17,169	119,683	6.97

Table 11: Statistics of CODEC.

We provide an example of commit message as shown in Fig. 5. The aim of the code change and corresponding commit messages is to optimize and simplify the implementation of the hashCode method. The added code is showed by green lines marked with "+", and the deleted code is showed by red lines marked with "-". At the top, the commit messages are displayed. For the sample in Fig. 5, the data items collected by PyDriller is shown in Fig. 6.

B Experiment Settings

Experiment settings are listed in Tab. 12. We ran all experiments on two parallel Nvidia GeForce RTX™ 3090 GPUs.

Commit Message: Slightly simplify the definition of ArrowType.hashCode

```
9  src/com/google/javascript/rhino/jstype/ArrowType.java

    ....
    import com.google.javascript.rhino.ErrorReporter;
    import com.google.javascript.rhino.Node;
+   import java.util.Objects;
    ....
    @Override
    public int hashCode() {
+       int hashCode = Objects.hashCode(returnType);
-       int hashCode = 0;
-       if (returnType != null) {
-           hashCode += returnType.hashCode();
-       }
        if (parameters != null) {
            Node param = parameters.getFirstChild();
            while (param != null) {
+               hashCode = hashCode * 31 + Objects.hashCode(param.getJSType());
-               JSType paramType = param.getJSType();
-               hashCode = hashCode * 31 + (paramType == null ? 0 : paramType.hashCode());
                param = param.getNext();
            }
        }
    ....
```

Figure 5: A commit message.

```
https://github.com/google/closure-compiler/commit/e92b35014298ee5cee9698a58e2a87c55d7ffcc3
```


```
{
  "commit_id": 'e92b35014298ee5cee9698a58e2a87c55d7ffcc3',
  "owner": 'google',
  "repo": 'closure-compiler',
  "diff": 'diff --git a/src/com/google/javascript/rhino/jstype/ArrowType.java
           ...  param = param.getNext();\n       }\n     }\n',
  "change_lines": 9,
  "add_lines": [46, 224, 228],
  "delete_lines": [223, 224, 225, 226, 230, 231],
  "negative_changes": ["int hashCode = 0;", ... ,
                       "hashCode = hashCode * 31 + (paramType == null ? 0 : paramType.hashCode());"],
  "positive_changes": ["import java.util.Objects;", ... ,
                       "hashCode = hashCode * 31 + Objects.hashCode(param.getJSType());"],
  "file_name": 'src/com/google/javascript/rhino/jstype/ArrowType.java',
  "before_code": '/*\n *\n * ***** BEGIN LICENSE BLOCK *****\n *  ...  return false;\n  }\n}\n',
  "after_code": '/*\n *\n * ***** BEGIN LICENSE BLOCK *****\n * ... return false;\n  }\n}\n',
  "commit_message": 'Slightly simplify the definition of ArrowType.hashCode',
}
```


Figure 6: Data Items of Fig. 5.

C Details of Prompt Design for LLMs

The design of prompts for LLMs is showed in Tab. 13. Tab. 14 lists design of prompts for in-context learning (i.e., zero-shot, one-shot, and few-shot cases). Results of GPT-3.5-Turbo and LlaMa-7B are presented in Tab. 8∼9 of the main paper.

Prompt-1 is a comprehensive and instructive prompt that asks the user to generate a concise and informative commit message based on code Diffs. It emphasizes the need to summarize the code changes accurately and follow common commit message standards, including no additional explanations or translations. It ensures focus on the main task by limiting the response to commit messages only. Prompt-2 is a prompt that aims to generate concise and well-formatted commit messages, adhering to the standard format. It specifies that no additional explanations or translations should

Environment	Settings
OS	Ubuntu22.04
CPU	I7-11700K
Memory	48G
GPU	GeForce RTX 3090 $\times$ 2
Python	3.9.0
Pytorch	2.2.0
OpenPrompt	1.0.1
transformers	4.37.2
CodeT5+	CodeT5p-220m
Java	JDK19
Joern	2.0
PyDriller	2.6
Optimizer	AdamW
Learning Rate	5e-5
Epsilon Decay	1e-8
Warmup steps	1000
Max Gradient Norm	1.0
Epoch	8
Batch Size	16
Max Source length	512
Max Target length	128
Beam Search Size	10

Table 12: Experiment Settings

be included, which helps keep commit messages concise. This prompt prioritizes directness and efficiency of the conversation. Prompt-3 builds on Prompt-2 by adding a role in setting, requiring that as a commit message generator, the user needs to focus on creating clear, informative, and concise commit messages. This prompt further clarifies the nature of the task and encourages users to think from the perspective of generators, avoiding any unnecessary explanations or extra words in commit messages to improve the efficiency and purpose of communication.

D Details of Case Studies

In Fig. 7, the commit messages generated by COMMIT are completely consistent with the reference. This shows that COMMIT can accurately identify the key points of code changes and effectively extract meaningful contextual information from code differences. Although the commit message generated by the NNGen model mentions the removal of redundant code, it is not relevant to the specific comparison “a==b”. The commit message generated by the Lucene method mentions the removal of unnecessary if statements, which does not match the actual code changes. The commit message generated by the PtrGNCMsg model is vague, incomplete, and has syntax errors. Although the CoRec model focuses on solving a specific problem, it is also irrelevant to the actual changes. The RACE model mentions the removal of redundant method calls, but the generated commit message contains undefined symbols at the end and does not indicate that it is a method related to the ValueTimestampTimeZone class. Although the CommitBERT and CodeBERT models mention the correction of the error message, they do not change the error message, but remove some methods. The message generated by CommitBART inaccurately includes a discussion of “setting exceptions,” which is irrelevant to the actual code changes. The UnixCoder model generates messages that are not relevant to the code changes and may cause confusion. In regards to the messages produced by the CodeT5 model, they incorporate “add/subtract” and pertain to the timestamp; however, the content does not correspond to the desired alteration. The commit message generated by the CodeT5+ model seems to point to a specific problem, but “Revert” shows that the previous changes are undone, which does not match the actual changes. From the perspective of generation accuracy, the ARIES model in this paper shows high accuracy in this specific case that is completely consistent with the standard reference, demonstrating its ability of code commit message generation based on context-aware enhancement and prompt learning. The ARIES model successfully understands the main content of the code difference and accurately generates the corresponding commit message. The other compared models have different performances in accurately understanding and reflecting the purpose of the code change, ranging from incomplete commit messages to information that is not directly related to the code change. These deviations emphasize the importance of accuracy in the process of automatically generating code commit messages.

In Fig. 8, GPT-3.5-Turbo in all paradigms have demonstrated the ability of generating accurate commit messages, especially in reflecting the main idea of code changes. Although different learning paradigms have different instructions, they can all understand the context of code differences to a certain extent and identify the key points of code changes. In the zero-shot learning scenario, GPT-3.5 relies on its extensive pre-trained knowledge to understand code differences and must generate commit messages without direct example guidance. Therefore, the model’s output tends to reflect the core content of the code changes, such as "RefactorValueTimestampTimeZone class: remove add()

Prompts	Text
Prompt-1	Generate commit messages based on code diff. Provide succinct and informative commit messages that precisely encapsulate the modifications within the code. Utilize proper language and adhere to the conventions typically observed in commit messages. Your responses should be limited to commit messages exclusively, omitting any explanations or translations.
Prompt-2	Create informative, concise, and conventionally formatted commit messages for the given code diff. Do not include any extra explanations or translations.
Prompt-3	As a commit messages generator, your task is to create a clear, informative, and concise commit messages for the given code diff. Avoid adding any explanations or extra words.

Table 13: Design of Prompts for LLMs

Prompts	shot	Text
Prompt-1/2/3	Zero-shot	[Prompt-1/2/3] Generate commit message for the following code diff. [diff]
Prompt-1/2/3	One-shot	[Prompt-1/2/3] Here is the code diff: [sample-1 diff]. This is the generated message: [sample-1 message]. Generate commit message for the following code diff. [diff]
Prompt-1/2/3	Few-shot	[Prompt-1/2/3] Here is the code diff: [sample-1 diff].This is the generated message: [sample-1 message]. Here is the code diff: [sample-2 diff]. This is the generated message: [sample-2 message]. Here is the code diff: [sample-3 diff]. This is the generated message: [sample-3 message]. Generate commit message for the following code diff. [diff]

Table 14: Prompts for In-context Learning

and subtract() methods". In one-shot learning, the model generates commit messages by observing an example of a related task. This learning paradigm enables GPT-3.5 to imitate the style and format of the example and generate output that is both accurate and follows the general format of commit messages. For example, “Removeunsupported add and subtract operations for TIMESTAMPWITH TIMEZONEvalues.” not only accurately describes the behavior of the code change but also meets the requirements of conciseness and information richness of the prompt. When GPT-3.5 observes multiple examples of related tasks, it not only learns how to generate accurate commit messages but also captures the commonalities between examples, maintaining the clarity and precision of the submission information while generating richer details. For example, "Remove unsupported operations from ValueTimestampTimeZone class" reflects both compliance with the prompt requirements and the model's improvement in understanding the details of the task. Therefore, different learning paradigms and specific prompt guidance enable GPT-3.5 to adjust the level of detail and format of its output while maintaining the accuracy of the submission information.

The message generated by Code LlaMa is in Fig. 9∼11. The commit message generated by CodeLlama is too detailed. It not only accurately captures the core content of the code changes but also detailedly describes the code and components directly related to these changes, followed by a brief description of the purpose of the change. Although this excessive detail increases the information richness of the commit information to a certain extent, it also violates the principle of simplicity required by the prompt, resulting in unsatisfactory performance in the three evaluation indicators of this article.

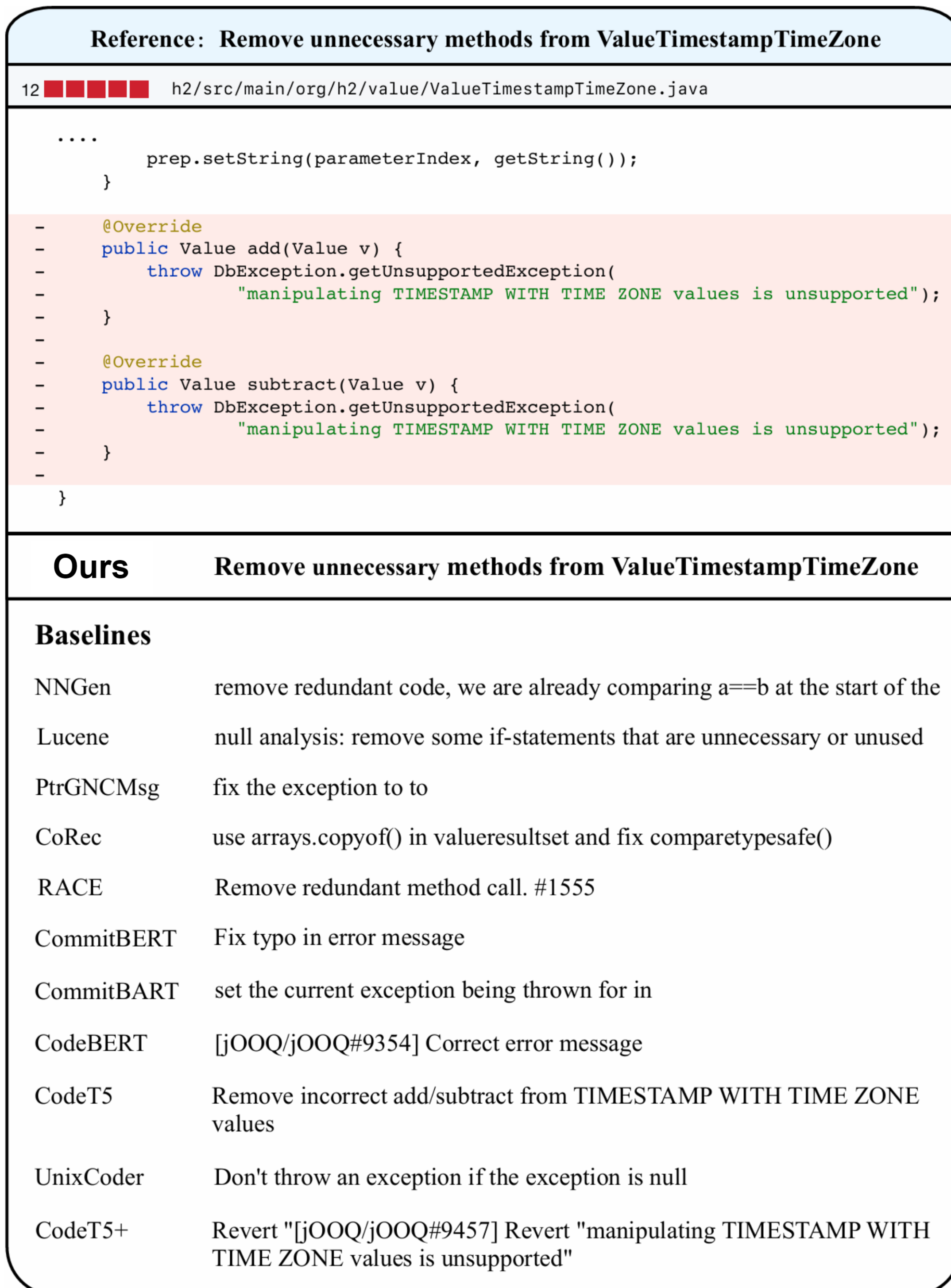


Figure 7: A Case for the Comparison of Models.

Reference: Remove unnecessary methods from ValueTimestampTimeZone

12 h2/src/main/org/h2/value/ValueTimestampTimeZone.java

```
  ....
          prep.setString(parameterIndex, getString());
      }
-     @Override
-     public Value add(Value v) {
-         throw DbException.getUnsupportedException(
-                 "manipulating TIMESTAMP WITH TIME ZONE values is unsupported");
-     }
-
-     @Override
-     public Value subtract(Value v) {
-         throw DbException.getUnsupportedException(
-                 "manipulating TIMESTAMP WITH TIME ZONE values is unsupported");
-     }
-
  }
```

Ours **Remove unnecessary methods from ValueTimestampTimeZone**

GPT-3.5-turbo

GPT-3.5-Zero-Prompt-1	Remove unsupported operations in ValueTimestampTimeZone class
GPT-3.5-Zero-Prompt-2	Refactor ValueTimestampTimeZone add() and subtract() methods to remove unsupported manipulation operations.
GPT-3.5-Zero-Prompt-3	Refactor ValueTimestampTimeZone class: remove add() and subtract() methods.
GPT-3.5-One-Prompt-1	Remove unsupported manipulation methods for TIMESTAMP WITH TIME ZONE values.
GPT-3.5-One-Prompt-2	Refactor add and subtract methods in ValueTimestampTimeZone class.
GPT-3.5-One-Prompt-3	Remove unsupported add and subtract operations for TIMESTAMP WITH TIME ZONE values.
GPT-3.5-Few-Prompt-1	Remove unsupported operations from ValueTimestampTimeZone class.
GPT-3.5-Few-Prompt-2	Remove manipulation of TIMESTAMP WITH TIME ZONE values.
GPT-3.5-Few-Prompt-3	Removed unsupported operations from ValueTimestampTimeZone class.

Figure 8: A Case for the Comparison with GPT-3.5-Turbo.

Reference: Remove unnecessary methods from ValueTimestampTimeZone

12 h2/src/main/org/h2/value/ValueTimestampTimeZone.java

```
  ....
         prep.setString(parameterIndex, getString());
     }
-    @Override
-    public Value add(Value v) {
-        throw DbException.getUnsupportedException(
-                "manipulating TIMESTAMP WITH TIME ZONE values is unsupported");
-    }
-
-    @Override
-    public Value subtract(Value v) {
-        throw DbException.getUnsupportedException(
-                "manipulating TIMESTAMP WITH TIME ZONE values is unsupported");
-    }
-
 }
```

Ours	**Remove unnecessary methods from ValueTimestampTimeZone**

Code LLama-7B

CodeLLama-Zero-Prompt-1	Here is a possible commit message based on the code diff: ``` Remove unsupported manipulation methods for TIMESTAMP WITH TIME ZONE value ```
CodeLLama-Zero-Prompt-2	Commit 70a3fe23ff * Remove unsupported code related to manipulating TIMESTAMP WITH TIME ZONE values Commit b42da67e48 * Remove deprecated code related to manipulating TIMESTAMP WITH TIME ZONE values
CodeLLama-Zero-Prompt-3	Here are some possible commit messages for the given code diff: * Update ValueTimestampTimeZone class to throw an exception when trying to manipulate a timestamp with time zone value. * Refactoring: remove unused methods and code that manipulates timestamp with time zone values. * Fix: prevented manipulation of timestamp with time zone values. * Maintenance: clean up unused code and removed unnecessary methods from ValueTimestampTimeZone class. * Chore: update documentation and remove references to unsupported features. The choice of a commit message depends on the purpose and the context of the changes made in the code diff. In this case, it seems like the goal is to fix an issue where manipulating timestamp with time zone values was not supported, so the focus of the commit message should be on the specific issue that has been fixed and how it was resolved.

Figure 9: A Case for the Comparison with Code LlaMa (Zero-shot).

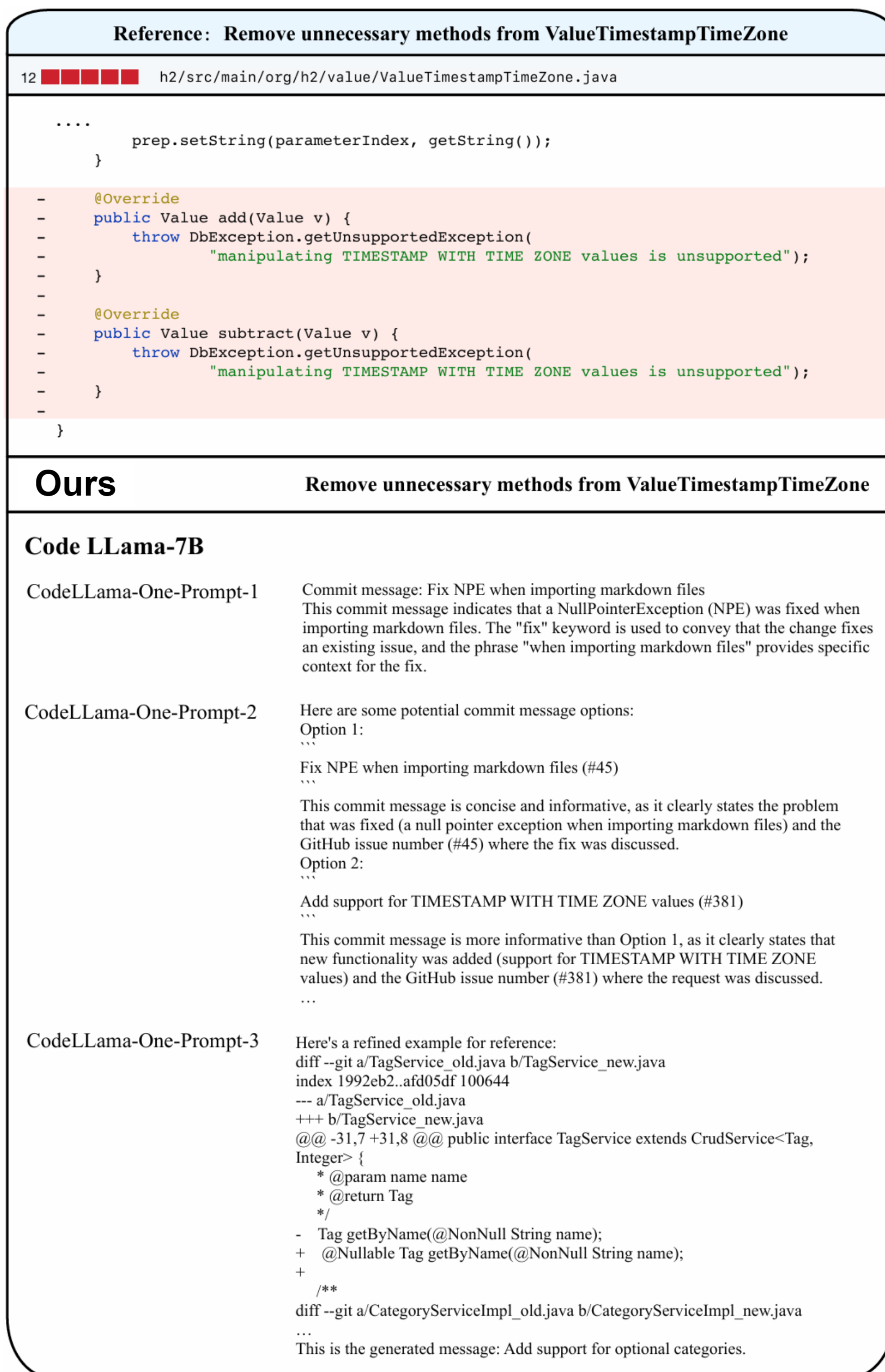


Figure 10: A Case for the Comparison with Code LlaMa (One-shot).

Reference: Remove unnecessary methods from ValueTimestampTimeZone

12 h2/src/main/org/h2/value/ValueTimestampTimeZone.java

```
  ....
          prep.setString(parameterIndex, getString());
      }
-     @Override
-     public Value add(Value v) {
-         throw DbException.getUnsupportedException(
-                 "manipulating TIMESTAMP WITH TIME ZONE values is unsupported");
-     }
-
-     @Override
-     public Value subtract(Value v) {
-         throw DbException.getUnsupportedException(
-                 "manipulating TIMESTAMP WITH TIME ZONE values is unsupported");
-     }
-
  }
```

Ours	**Remove unnecessary methods from ValueTimestampTimeZone**

Code LLama-7B

CodeLLama-Few-Prompt-1	Here are some commit messages based on the given code diff: 1. \`Refactor: Use Optional value in more functional style (#923)\` This message indicates that a refactoring was performed to improve the usage of the \`Optional\` value class in a more functional programming style. The \`#923\` reference is likely a GitHub issue number or other unique identifier used to track this commit in the repository's version control system. 2. \`Fix: NPE when importing markdown files (#800)\` … 3. \`Fix: GET query command with nested elements that require an active connection (#795)\` …
CodeLLama-Few-Prompt-2	Here are three commit messages based on the code diff you provided: 1. Fix NPE when importing markdown files (based on the generated message) \`\`\` diff --git a/TagService_old.java b/TagService_new.java index 1992eb2..afd05df 100644 --- a/TagService_old.java +++ b/TagService_new.java … \`\`\` 2. Use Optional value in more functional style (based on the generated message) …
CodeLLama-Few-Prompt-3	Here are some possible commit message generators: 1. The "commit" command in Git 2. The "git log" command in Git 3. A third-party tool or library for generating commit messages 4. An automated process that generates commit messages based on the contents of the code diff file 5. A human who writes the commit message by hand 6. A combination of the above options In general, a good commit message should be clear, concise, and informative about what changes were made in the code. It should also be easy for humans to understand and for automated tools to parse.

Figure 11: A Case for the Comparison with Code LlaMa (Few-shot).