HierDiff: Enhancing Large Language Models for Commit Message Generation with Hierarchical Diff Information

Jinrui Sun, Tong Jia*, Minghua He, Ying Li
Peking University
jrsun25@stu.pku.edu.cn, jia.tong@pku.edu.cn,
hemh2120@stu.pku.edu.cn, li.ying@pku.edu.cn
*Corresponding author

Abstract

Commit messages are vital artifacts that summarize changes to facilitate software maintenance, debugging, and evolution tracking. However, existing Commit Message Generation (CMG) methods rely solely on code diffs and overlook higher-level architectural dependency diff information. To address this limitation, we propose HierDiff, a framework using architectural diffs to complement code diffs for improved commit generation. Through this framework, we further construct HierCMD, a new dataset that incorporates architectural diffs alongside much longer code diffs. Building on this resource, we develop HierDiff-6.7B, a lightweight CMG model trained via multi-stage hierarchical diff learning. To enable more rigorous evaluation, we additionally extend the existing MCMD-New-Test benchmark by incorporating architectural diff information, resulting in MCMD-New-TestX. Experimental results on MCMD-New-TestX and HierCMD demonstrate that HierDiff-6.7B achieves state-of-the-art performance in CMG, even surpassing the renowned closed-source model GPT-4o by 2.1% to 29.8% across four evaluation metrics. We release the data and code at: https://github.com/mianmaner/HierDiff.

1 Introduction

In modern software development, commit messages serve as an essential communication medium between developers and the evolving codebase, helping practitioners quickly understand code changes without deeply inspecting modified files. Prior studies have shown that high-quality commit messages facilitate software maintenance, debugging, code comprehension, and code review (Mockus and Votta, 2000; Tao et al., 2012). However, under rapid development cycles, manually writing concise and informative commit messages for large-scale changes remains challenging and time-consuming (Tao et al., 2021; Dyer et al., 2013; Tian et al., 2022). Poor commit messages not only reduce documentation quality but are also associated with higher software defect proneness (Li and Ahmed, 2023). Consequently, automated commit message generation has attracted increasing attention. In recent years, researchers have developed a wide range of commit message generation (CMG) techniques, which can be broadly categorized into retrieval-based (Liu et al., 2018; Huang et al., 2020, 2017), translation-based (Loyola et al., 2017; Nie et al., 2021; Xu et al., 2019; Dong et al., 2022; Liu et al., 2019), and hybrid approaches (He et al., 2023; Liu et al., 2022; Shi et al., 2022; Wang et al., 2021). Although these methods can generate syntactically plausible commit messages, they often rely on shallow change patterns, leading to limited informativeness and poor readability. More recent studies (Wu et al., 2025; Jiang et al., 2024) have introduced Large Language Models (LLMs) to improve generation quality. However, existing LLM-based CMG approaches still depend primarily on code diffs as input, while overlooking higher-level architectural change information. In practice, developers compose commit messages based on a holistic understanding of the software system,

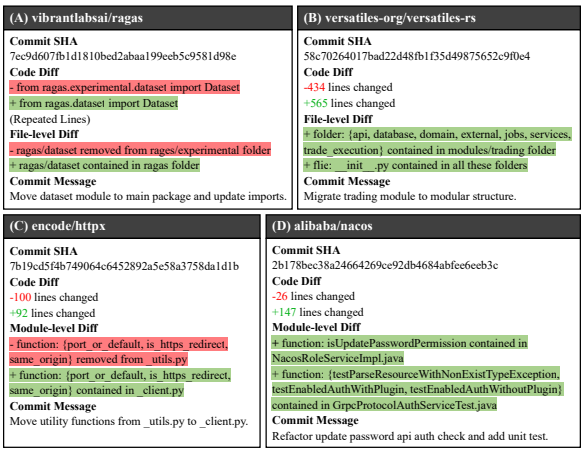

Figure 1: Four real-world GitHub commit examples.

Findings of the Association for Computational Linguistics: EMNLP 2026, pages 9466–9480
October 24-29, 2026 ©2026 Association for Computational Linguistics

including inter-file dependencies, module responsibilities, and cross-module interaction semantics.

Moreover, when code changes are large in scale, such as during major refactoring or module migration, statement-level code diffs often become excessively verbose. In these scenarios, the essence of the change is usually more clearly reflected in file-level or module-level dependency evolution. Prior studies have shown that commits involving architectural changes are often described based on module-level or even file-level dependency diffs rather than purely relying on fine-grained code diffs (Motta et al., 2018; Cui et al., 2021). Figure 1 illustrates four real-world commits from GitHub where architectural changes are explicitly reflected in commit messages. Examples (A) and (B) illustrate file-level diffs, while (C) and (D) illustrate module-level diffs.

To address this, we propose HierDiff, a framework spanning the entire pipeline from data construction to model training. HierDiff introduces a Hierarchical Dependency Graph (HDG) to model the dependency architecture of a codebase at each revision and extracts hierarchical diffs by tracking HDG evolution across commits. These diffs compactly capture architectural dependency changes. To bridge the semantic gap between architectural and code diffs, we further propose Staged Diff Learning, a multi-stage fine-tuning strategy that progressively guides the model from architectural diffs to code diffs and finally to their combination, enabling more effective joint representation learning and substantially improving commit message generation quality. Since CMG inherently involves source code modifications, directly invoking external LLM APIs raises privacy concerns for both individual developers and enterprises. To support privacy-preserving local deployment, we construct HierCMD, a new dataset containing both architectural-level diffs and substantially longer code diffs. Based on this dataset, we develop HierDiff-6.7B, a local LLM for CMG trained with the staged diff learning paradigm, enabling privacy-preserving commit message generation.

To enable more rigorous evaluation, we augment the public MCMD-New test set with architectural diffs, yielding MCMD-New-TestX. Extensive experiments on both MCMD-New-TestX and HierCMD demonstrate that HierDiff-6.7B achieves state-of-the-art performance on the CMG task, even outperforming strong closed-source LLMs such as GPT-4o and Gemini 2.5 Flash. In summary, our contributions are as follows.

- We are the first to incorporate architectural change information into the CMG task. We propose HierDiff, a framework that models architectural changes through a Hierarchical Dependency Graph and introduces Staged Diff Learning to effectively learn from architectural diffs across the full pipeline from data construction to model training.
- We construct HierCMD, a new CMG dataset containing architectural diffs and larger-scale code diffs that better reflect real-world commits. We also augment the public MCMD-New test set with architectural diff information, yielding MCMD-New-TestX, to support future CMG research on architectural changes.
- We develop HierDiff-6.7B, a local LLM specialized for commit message generation. HierDiff-6.7B outperforms existing open-source code LLMs and achieves state-of-the-art results, surpassing strong closed-source models such as GPT-4o and Gemini-2.5.

2 Background & Related Work

2.1 Architectural diffs in Commit Messages

Recent empirical studies underscore the prevalence and significance of architectural changes in commits (Motta et al., 2018; Cui et al., 2021). A substantial portion of commits involves architectural modifications such as module decomposition and migration, dependency restructuring, and cross-component interaction changes. In these cases, developers tend to describe changes at the architectural or module level rather than through fine-grained, statement-level code modifications. Moreover, empirical evidence (Cui et al., 2021) suggests that neglecting architectural changes during code review and evolution analysis can lead to architectural erosion and long-term design degradation. These findings indicate that architectural changes encode critical semantic signals about developer intent that low-level code diffs alone cannot fully capture. Therefore, effectively modeling architectural information is essential for both empirical software engineering research and automated commit message generation. Ignoring architectural changes yields an incomplete representation of change intent, which in turn limits the effectiveness and practical applicability of existing CMG approaches.

2.2 Commit Message Generation

Commit message generation aims to automatically generate concise descriptions of code changes. Existing methods mainly fall into four categories. Template-based methods (Buse and Weimer, 2010; Linares-Vásquez et al., 2015; Shen et al., 2016) rely on predefined structures but struggle with complex changes and high-level intent. Retrieval-based methods (Liu et al., 2018; Huang et al., 2020, 2017), such as NNGen (Liu et al., 2018), reuse messages from similar historical commits, yet their performance drops when similar examples are unavailable or noisy. Translation-based methods (Jiang et al., 2017; Loyola et al., 2017; Nie et al., 2021; Dong et al., 2022) formulate CMG as a sequence-to-sequence task; for instance, PtrGNCMsg (Liu et al., 2019) employs a pointer-generator mechanism to address OOV identifiers, though generated messages often lack readability and diversity. Hybrid methods (He et al., 2023; Liu et al., 2022; Shi et al., 2022) combine retrieval and generation, but insufficient integration between the two can limit performance. With LLMs demonstrating exceptional performance across various software engineering tasks (Sun et al., 2026; Tao et al., 2024; He et al., 2024, 2025b), their integration has further propelled the advancement of CMG through stronger reasoning and in-context learning capabilities. (Wu et al., 2025; Jiang et al., 2024). However, existing methods still rely mainly on code diffs and local statement-level context, overlooking higher-level architectural diff information. Our work addresses this limitation by incorporating architectural diff signals into LLM-based commit message generation.

3 Methodology

In this section, we propose HierDiff, a framework based on hierarchical diffs that effectively leverages architectural diffs as complementary knowledge to code diffs, covering the entire pipeline from data collection to model training. The overall framework of HierDiff is shown in Figure 2.

3.1 Hierarchical Dependency Graph

3.1.1 Graph Definition

As the codebase architecture continuously evolves along with commits, we first design a Hierarchical Dependency Graph (HDG) to explicitly model the dependency structure of the codebase at each revision. The HDG is formulated as a heterogeneous directed graph $G = (V, E)$, where $V = \{v_i\}_{i=1}^{n}$ denotes the node set and $E \subseteq V \times V$ denotes the edge set. Each node $v \in V$ is associated with a node-type mapping function $\tau(v) : V \to A$, where $A = \{\textit{folder}, \textit{file}, \textit{class}, \textit{function}\}$ represents different types of architectural entities. Similarly, each directed edge $e \in E$ is associated with a relation-type mapping function $\phi(e) : E \to R$, where $R = \{\textit{contain}, \textit{import}, \textit{invoke}, \textit{inherit}\}$ characterizes the dependency relationships between entities.

Based on the granularity of the involved entities, we further categorize dependencies into two levels to facilitate analysis: file-level and module-level. Dependencies that involve file-system entities, such as folders or files, are classified as file-level dependencies, whereas dependencies that only involve code entities, such as classes and functions, are regarded as module-level dependencies.

3.1.2 Graph Evolution

The initial HDG is built using the static parser Tree-Sitter (Brunsfeld et al., 2025) to extract structural information such as module definitions and inter-file or inter-module dependencies. However, in continuously evolving projects, the codebase is updated incrementally through commits. As the project scales, repeatedly reanalyzing the entire codebase from scratch after each update becomes increasingly time-consuming and impractical. To address this, we propose a commit-based update algorithm for HDG maintenance, with pseudocode given in Algorithm 1. Rather than reconstructing the HDG after each commit, our approach incrementally updates the graph by confining static analysis to only the files affected by the commit.

Given a commit C that modifies a set of files $\{f_1, f_2, \ldots, f_n\}$, the update proceeds as follows. First, for each modified file f, we handle file-system-level changes such as file addition, deletion, renaming, and movement. These operations directly affect node existence and identity in the HDG, and we update the graph accordingly to reflect newly introduced nodes, removed nodes, or updated file paths. Second, for files that are modified, we process code-level changes via static analysis. Specifically, we apply the static parser to the pre-commit and post-commit versions of file f, extracting the corresponding sets of modules and their dependency relations, denoted as $(nodes_{old}, edges_{old})$ and $(nodes_{new}, edges_{new})$, respectively. By comparing these two representations, we explicitly identify both node-level and

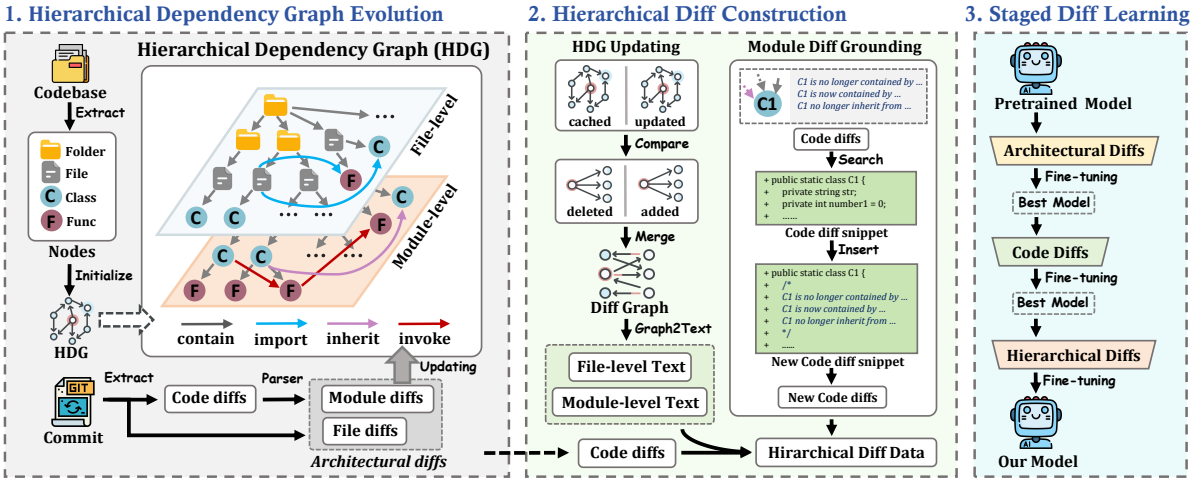

Figure 2: The overall framework of HierDiff.

dependency-level changes.

Notably, every node in the HDG is uniquely identified by a tuple of attributes comprising its name, type, and relative file path. These attributes enable reliable indexing and matching across commits, allowing all affected dependencies to be accurately captured through static analysis restricted solely to the modified files. Consequently, the proposed dynamic update algorithm avoids redundant global analysis while preserving correctness, enabling efficient HDG maintenance in evolving software projects.

Algorithm 1: Update algorithm for HDG

Input: HDG $G = (V, E)$, commit C with modified files $\{f_1, f_2, \ldots, f_n\}$
Output: Updated HDG G

foreach $f \in C$ **do**
  `// FileSys-level update`
  $c_f \leftarrow$ FILESYSDIFF(f);
  UPDATE(G, c_f);
  `// Code-level update`
  `//` N, E `denote nodes and edges`
  if f *is modified* **then**
    $(N_{old}, E_{old}) \leftarrow$ EXTRACT(f_{old});
    $(N_{new}, E_{new}) \leftarrow$ EXTRACT(f_{new});
    $c_c \leftarrow$ CODEDIFF$(N_{old}, N_{new}, E_{old}, E_{new})$;
    UPDATE(G, c_c);
return G;

3.2 Hierarchical Diff Construction

When the HDG is updated, we compare its states before and after the update to derive a dependency change graph ΔG that explicitly captures all architectural diffs introduced by a commit. Formally, ΔG consists of all added and removed nodes, and all added and removed edges with their source and target nodes. Each node and edge is annotated with its edit operation (`ADD` or `DEL`), yielding an unambiguous representation of architectural evolution.

Although ΔG precisely characterizes architectural changes, LLMs are pretrained on sequential text and struggle to directly consume structured graph representations. To bridge this gap, inspired by prior work on graph-to-text encoding (Fatemi et al., 2023; He et al., 2025a), we transform ΔG into textual representations that preserve structural semantics while remaining compatible with LLM processing. As illustrated in Figure 2, we adopt distinct encoding strategies for file-level and module-level diffs. For file-level diffs, we assign unique identifiers to all nodes in ΔG and serialize added and removed nodes and edges using a predefined textual format that records node identities, dependency directions, and edit operations. For module-level diffs, we generate node-centric textual descriptions that separately summarize the added and removed dependency edges associated with each module. This localized representation allows module-level diff texts to be naturally inserted into relevant code contexts without disrupting code structure. Following this encoding, the graph edits are transformed into natural language descriptions that retain dependency semantics. Prior work (Fatemi et al., 2023; He et al.,

2025a) has shown that such representations substantially improve LLMs' ability to comprehend and leverage structured information.

Since module-level dependencies are tightly coupled with code changes, for each module-level node in ΔG, HierDiff identifies the corresponding functions or classes in the code diffs and inserts the encoded diff text as annotations at the relevant locations. These annotations enable the LLM to map local code diffs to higher-level architectural diffs. Notably, each encoded module-level diff text contains complete bidirectional dependency change information, so inserting it at either the source or target node incurs no information loss.

3.3 Staged Diff Learning

Despite the necessity of local LLM deployment for data confidentiality (Rabin et al., 2025; Shanmugarasa et al., 2025), we propose Staged Diff Learning, a progressive fine-tuning strategy inspired by curriculum learning (Soviany et al., 2022).

The core intuition is twofold. First, architectural diffs provide abstract, high-level semantic signals with relatively low noise, while code diffs are information-dense and noisy, naturally motivating a curriculum-style progression from coarse-grained structural understanding to fine-grained implementation details. Second, developers typically possess prior architectural knowledge when composing commit messages; to mimic this process, we enable LLMs to internalize architectural semantics before reasoning over code-level changes. Accordingly, HierDiff performs multi-stage LoRA-based fine-tuning (Hu et al., 2021) on DeepSeek-Coder-6.7B-Instruct to obtain HierDiff-6.7B. Staged Diff Learning consists of three stages, each targeting a distinct granularity of change information with stage-specific prompts, which are provided in Appendix C. Specifically, the model is first trained on architectural diffs to capture high-level structural modifications and their motivations, then on code diffs to learn precise implementation-level semantics such as logic updates and bug fixes, and finally on hierarchical diffs combining both information sources, allowing the model to align architectural evolution with code-level changes and generate commit messages that are both structurally informative and semantically precise.

4 HierCMD Dataset

Since existing datasets lack architectural change information, we construct HierCMD, a new dataset tailored for our task. This section details the construction process and provides a comparative analysis with prior datasets.

4.1 Dataset Construction

Following prior work (Liu et al., 2018; Xu et al., 2019; Liu et al., 2019, 2022; Tao et al., 2021; Jiang et al., 2024), we use PyDriller (Spadini et al., 2018) to collect popular Java and Python repositories from GitHub, each with at least 1,000 stars. We select 60 projects of varying sizes. For each commit, in addition to code diffs, we extract a diff graph via the HierDiff framework to represent architectural diffs. In total, the dataset comprises approximately 59.9K Java commits and 69.5K Python commits.

We strictly follow the filtering steps in prior work to mitigate low-quality commits (Liu et al., 2018; Xu et al., 2019; Liu et al., 2022; Tao et al., 2021; Jiang et al., 2024; Wu et al., 2025), including basic message quality control and linguistic pattern-based filtering, ensuring the overall reliability of reference commit messages. A key distinction from existing datasets is that all existing CMG datasets were originally built for NMT-based approaches and retain only very short code diffs, with most limiting diff length to 100 tokens. This restriction no longer reflects real-world development scenarios. As shown in Table 1, the majority of commits in our collected data exhibit code diffs longer than 100 tokens. Moreover, with the expanded context windows of modern LLMs (Wang et al., 2024), larger diffs can be effectively utilized. We therefore adopt a 2K token threshold and retain all commits whose code diffs fall below this limit.

After filtering and deduplication, HierCMD retains 7.8K Java commits and 7.2K Python commits. Following prior splitting conventions (Liu et al., 2018; Xu et al., 2019; Liu et al., 2019, 2022; Tao et al., 2021; Jiang et al., 2024), we randomly partition HierCMD into training, validation, and test sets with an 80%, 10%, and 10% ratio, respectively.

4.2 Comparison

We compare HierCMD with several existing commit message generation datasets (Jiang et al., 2017; Liu et al., 2018; Xu et al., 2019; Liu et al., 2019; Loyola et al., 2017; Liu et al., 2022; Tao et al., 2021; Jiang et al., 2024; Wu et al., 2025) in Ta-

Metric	Range	Java	Python
Diff Length	≤100	15,343 (25.6%)	29,321 (42.2%)
	>100	44,542 (74.4%)	40,160 (57.8%)
Tokens	≤2048	53,707 (89.7%)	65,093 (93.7%)
	>2048	6,178 (10.3%)	4,388 (6.3%)

Table 1: Statistics of commit diff length and tokens.

ble 2 and highlight three main contributions of our dataset. First, it captures previously overlooked architectural diff information in the form of diff graphs via HDG modeling, laying the foundation for CMG research that leverages architectural information. Second, it expands the code diff scale from a maximum of 100 tokens to 2K tokens, accommodating the context windows of most modern LLMs. As shown in Table 1, this expansion increases the coverage of real-world code diffs from less than half to approximately 90%, demonstrating HierCMD's comprehensive coverage and practical utility. Third, we observed that none of the existing datasets provides source code for data collection and filtering, forcing researchers to analyze filtered data on a case-by-case basis when extending these datasets. To address this, we have open-sourced the full pipeline for automated dataset construction, allowing extensions to be integrated at any stage of collection or filtering, which substantially enhances the extensibility of HierCMD. Further introduction of these datasets is provided in the Appendix D.

5 Experiments

5.1 Datasets

We conducted experiments on the widely used public CMG dataset MCMD-New (Wu et al., 2025) and HierCMD to evaluate the effectiveness of HierDiff-6.7B. Since all existing datasets, including MCMD-New, lack architectural diff information and thus cannot support our approach, we applied the HierDiff framework to augment the MCMD-New test set. Specifically, based on commit SHA information, we used static tools (Spadini et al., 2018; Brunsfeld et al., 2025) to analyze the current and previous codebases of each commit in the dataset, extracting the HDG and comparing them to obtain the diff graph. Commits for which the codebase could not be found were omitted. This process produced a test set with architectural diff information, which we refer to as MCMD-New-TestX, containing a total of 3,457 commits.

5.2 Metrics

We adopt four evaluation metrics that are widely used in prior studies (Liu et al., 2022; Shi et al., 2022; He et al., 2023; Wu et al., 2025), including BLEU (Papineni et al., 2002), METEOR (Banerjee and Lavie, 2005), ROUGE-L (Lin, 2004), and CIDEr (Vedantam et al., 2015). These metrics are prevalent metrics in machine translation, text summarization, and image captioning. In addition, to enable more comprehensive evaluation, we further conduct both human and LLM-based evaluation on the *what* and *why* dimensions. Detailed descriptions of all metrics are provided in Appendix B.

5.3 Baselines

Since we extend the code diff length to up to 2K tokens to better accommodate LLMs, traditional CMG models with limited input capacities are no longer applicable. Therefore, we select only RACE (Shi et al., 2022) and COMMIT (Jiang et al., 2024), which are built upon pretrained language model encoders, as baselines. Furthermore, the state-of-the-art CMG method, OMG (Li et al., 2024), requires costly expert annotations for all data (only 381 commits were sampled for evaluation in their work), and is thus excluded from our baselines. Following related works (Wu et al., 2025; He et al., 2025a), to assess the effectiveness of HierDiff-6.7B, we finally compare it against state-of-the-art LLMs. Specifically, we adopt three advanced closed-source models as baselines, including GPT-3.5-Turbo, GPT-4o, and Gemini-2.5-Flash (Brown et al., 2020; OpenAI et al., 2024; Comanici et al., 2025). In addition, we select five representative open-source code summarization LLMs as baselines: DeepSeek-Coder-6.7B-Instruct (DeepSeek-AI et al., 2024), CodeLlama-7B (Rozière et al., 2023), Magicoder-S-DS-6.7B (Wei et al., 2024), Qwen2.5-Coder-7B-Instruct (Hui et al., 2024), and WaveCoder-Ultra-6.7B (Yu et al., 2024).

5.4 Implementation Details

To ensure reproducibility, all experiments were conducted on a server with 32 CPU cores, 240 GB RAM, and two RTX 4090 GPUs. Detailed implementation settings and hyperparameters are provided in Appendix B.

5.5 Evaluation Results

Our experiments are primarily designed to investigate the following research questions:

Dataset	#Commits	Availability	Reproducible	Deduplicated	Filtered	Code Diff	Architectural Diff
NNGen	27,144	✗	✗	✓	✓	length $\leq$ 100	✗
CoDiSum	90,661	✗	✗	✗	✗	length $\leq$ 200	✗
PtrGNCMsg	32,663	✗	✗	✓	✓	length $\leq$ 100	✗
MultiLang	153,444	✓	✗	✓	✗	length $\leq$ 100	✗
ATOM	197,968	✗	✗	✓	✓	length $\leq$ 100	✗
MCMD	2,250,000	✓	✗	✓	✓	length $\leq$ 100	✗
CODEC	17,169	✓	✗	✓	✓	lines $\leq$ 20	✗
MCMD-New	229,492	✗	✗	✓	✓	length $\leq$ 100	✗
HierCMD	15,008	✓	✓	✓	✓	**tokens $\leq$ 2,048**	✓

Table 2: Comparison of HierCMD with exsting datasets.

- **RQ1**: How effective is HierDiff-6.7B?
- **RQ2**: How effective is Staged Diff Learning?
- **RQ3**: Can HierDiff-6.7B generate commit messages using only architectural diffs?
- **RQ4**: How does HierDiff-6.7B perform under human and LLM-based evaluation?

5.5.1 RQ1: How effective is HierDiff-6.7B?

We evaluate HierDiff-6.7B and all baselines on the HierCMD and MCMD-New-TestX datasets, with results shown in Table 3. Overall, HierDiff-6.7B consistently achieves the best or highly competitive performance across both datasets, outperforming existing open-source CMG models and code LLMs. Compared with the strongest open-source baseline, HierDiff-6.7B improves all four metrics on both datasets, demonstrating its ability to generate accurate and informative commit messages. It also outperforms strong closed-source models such as GPT-4o and Gemini-2.5-Flash on several key metrics. On HierCMD, HierDiff-6.7B achieves the best performance across all four metrics, improving over GPT-4o by 2.1%–29.8%. On MCMD-New-TestX, HierDiff-6.7B achieves the highest ROUGE-L and CIDEr scores, while remaining highly competitive on BLEU and METEOR.

To further examine the contribution of the hierarchical diff representation, we additionally evaluate the prompt-based LLMs with two input settings: Code Diff and Hierarchical Diff. As shown in Table 3, incorporating hierarchical information leads to only marginal changes for the general-purpose LLMs. For GPT-4o, the hierarchical input slightly improves METEOR and CIDEr on HierCMD, while producing comparable results on MCMD-New-TestX. Similarly, Gemini-2.5-Flash obtains small gains in BLEU, METEOR, ROUGE-L, and CIDEr on MCMD-New-TestX, although its HierCMD performance varies slightly across metrics. In contrast, HierDiff-6.7B achieves substantially stronger gains under the same hierarchical input setting, suggesting that its task-specific staged training enables it to more effectively exploit the structural information captured by the HDG-derived dependency diffs.

5.5.2 RQ2: How effective is Staged Diff Learning?

To comprehensively evaluate Staged Diff Learning, we conduct two sets of experiments: (1) analyzing model performance across different training stages, and (2) comparing staged training with direct fine-tuning under diff ablation settings. The results are shown in Figure 3 and Figure 4. Overall, Stage 1 already provides clear improvements over the base model, indicating that architectural diffs offer useful high-level semantic guidance. Introducing code diffs in Stage 2 brings the largest performance gains, showing that fine-grained implementation details are the primary source of improvement. Stage 3 further yields consistent incremental benefits by jointly aligning architectural and code-level information. In the ablation study, Staged Diff Learning consistently outperforms direct fine-tuning on hierarchical diffs across both datasets, while direct fine-tuning with hierarchical diffs also performs better than using only code diffs. These results demonstrate that progressively learning from architectural diffs, code diffs, and their joint representations enables more effective alignment and superior commit message generation quality.

5.6 RQ3: Can HierDiff-6.7B generate commit messages with only architectural diff?

In real-world software development, some commits either contain code diffs exceeding the model input limit or involve only file-level changes with-

Type	Model	MCMD-New-TestX				HierCMD			
		BLEU	METEOR	ROUGE-L	CIDEr	BLEU	METEOR	ROUGE-L	CIDEr
CMG Models	RACE	5.14	7.12	7.01	0.13	5.25	7.27	9.22	0.18
	COMMIT	10.21	9.66	10.91	0.31	10.87	9.02	11.53	0.35
Code PLMs	CodeLlama	5.38	7.45	7.91	0.12	4.36	6.46	9.38	0.11
	WaveCoder	7.74	7.03	9.49	0.21	6.52	7.13	10.63	0.17
	Magicoder	8.81	9.03	9.55	0.24	7.48	7.84	11.01	0.23
	DeepSeek-Coder	8.20	7.68	8.27	0.21	7.08	6.74	9.90	0.22
	Qwen-2.5-Coder	7.28	7.51	10.18	0.21	9.23	10.08	15.94	0.34
General LLMs	GPT-3.5-Turbo (Code Diff)	11.47	9.52	13.48	0.38	10.89	8.64	15.22	0.40
	GPT-3.5-Turbo (Hier. Diff)	11.31	9.44	13.53	0.37	10.89	8.28	15.03	0.34
	GPT-4o (Code Diff)	11.85	10.57	15.78	0.45	12.62	11.09	19.96	0.57
	GPT-4o (Hier. Diff)	11.90	10.58	15.70	0.44	12.56	11.20	19.83	0.62
	Gemini-2.5-Flash (Code Diff)	14.04	11.48	16.04	0.48	14.36	11.29	18.22	0.51
	Gemini-2.5-Flash (Hier. Diff)	**14.10**	**11.50**	16.59	0.50	14.27	11.31	19.42	0.59
Ours	**HierDiff-6.7B**	13.22	10.41	**18.61**	**0.56**	**16.01**	**11.32**	**23.36**	**0.74**

Table 3: Performance comparison on MCMD-New-TestX and HierCMD. The best results are in bold. For general LLMs, we evaluate two input formats: Code Diff and Hierarchical Diff.

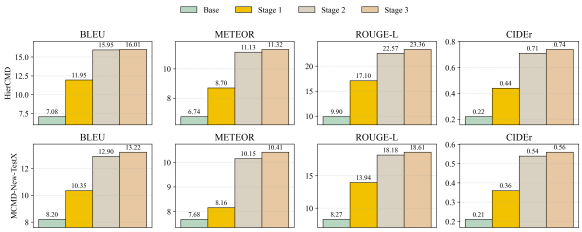

Figure 3: Performance across different training stages.

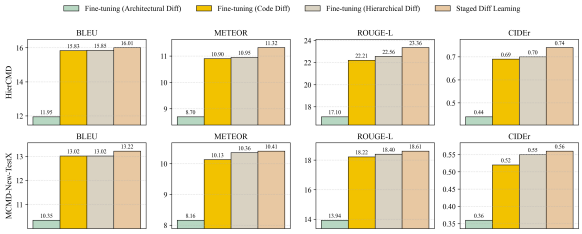

Figure 4: Performance compared to direct fine-tuning.

Model	BLEU	METEOR	ROUGE-L	CIDEr
GPT-3.5-Turbo	3.95	0.79	1.84	0.01
GPT-4o	2.36	2.01	2.22	0.04
Gemini-2.5	4.01	1.53	2.18	0.03
HierDiff-6.7B	**5.88**	**3.60**	**7.90**	**0.08**

Table 4: Performance comparison on MCMD-New-TestX using only architectural diffs.

Model	BLEU	METEOR	ROUGE-L	CIDEr
GPT-3.5-Turbo	4.05	0.84	2.11	0.01
GPT-4o	3.32	2.14	3.50	0.05
Gemini-2.5	4.39	2.24	3.28	0.05
HierDiff-6.7B	**7.14**	**3.91**	**9.85**	**0.16**

Table 5: Performance comparison on HierCMD using only architectural diffs.

out explicit code modifications, making existing CMG approaches inapplicable. By introducing architectural diffs, our framework naturally handles these scenarios. To further evaluate their effectiveness, we conduct experiments using only architectural diffs as input on HierCMD and MCMD-New-TestX, comparing HierDiff-6.7B with advanced general LLMs. As shown in Table 4 and Table 5, HierDiff-6.7B consistently outperforms all baselines across both datasets. On HierCMD, HierDiff-6.7B improves the strongest baseline by 62.6% in BLEU and 220.0% in CIDEr, while on MCMD-New-TestX it achieves gains of 46.6% in BLEU and 255.9% in ROUGE-L. These results demonstrate that HierDiff-6.7B can effectively capture high-level semantic intent from coarse-grained architectural changes, even without code-level information. More importantly, the performance of HierDiff-6.7B using only architectural diffs is already competitive with CodeLlama using full code diffs, highlighting the practical value of architectural diffs for commits where code changes are empty or exceed the model's input length limit.

5.7 RQ4: How does HierDiff-6.7B perform under human and LLM-based evaluation?

Following prior work (Zeng et al., 2025), we conduct human and LLM-based evaluations on the *what* and *why* dimensions using a 5-point Likert scale, complementing automatic metrics. The overall score is the equally weighted average of the two dimensions. Detailed settings are provided in Appendix B. Due to the high annotation cost, we randomly sample 300 cases from MCMD-New-TestX. As shown in Figure 5, HierDiff-6.7B outper-

forms GPT-4o but falls behind Gemini-2.5-Flash on the overall *what* and *why* evaluation, as general LLMs tend to generate more detailed but less concise messages (Briakou et al., 2024). Nevertheless, HierDiff-6.7B remains competitive in human-perceived generation quality.

To further assess other practical aspects of commit messages, we additionally evaluate *conciseness*, *specificity*, *style consistency*, and *hallucination* using the same 5-point scale. As shown in Table 6, HierDiff-6.7B achieves the highest scores in *conciseness* and *style consistency*, outperforming both GPT-4o and Gemini-2.5-Flash. GPT-4o performs best in *specificity*, while Gemini-2.5-Flash achieves the highest scores in *what* and *hallucination*. These results suggest a clear trade-off between informativeness and conciseness: proprietary LLMs tend to produce more detailed messages, whereas HierDiff-6.7B is better optimized for concise and stylistically consistent commit messages. Overall, the complementary human and LLM-based evaluations demonstrate that HierDiff-6.7B produces practically useful commit messages with competitive quality across multiple dimensions.

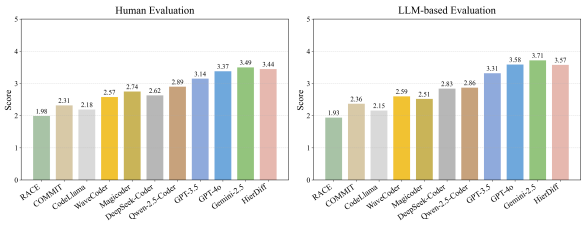

Figure 5: Performance across different training stages.

Dimension	HierDiff-6.7B	GPT-4o	Gemini-2.5-Flash
What	3.79	3.83	**3.95**
Why	3.35	3.33	**3.47**
Conciseness	**4.33**	2.77	4.09
Specificity	2.94	**4.00**	3.75
Style consistency	**3.71**	2.33	3.41
Hallucination	4.16	3.84	**4.52**

Table 6: LLM-based evaluation on additional dimensions on MCMD-New-TestX.

6 Conclusion

We propose HierDiff, a framework that models architectural changes via a Hierarchical Dependency Graph and employs a Staged Diff Learning strategy to capture architectural diffs throughout the pipeline. Using this framework, we construct HierCMD, a new CMG dataset that includes previously overlooked architectural diffs and retains commits with large-scale code diffs. Based on this, we develop HierDiff-6.7B, a large language model for commit message generation that achieves state-of-the-art performance, significantly outperforming all existing open-source code LLMs. Notably, it surpasses GPT-4o by 2.1%–29.8% across four metrics and outperforms Gemini-2.5-Flash by 16.0% and 16.7% on two key metrics. These results show that architectural diffs serve as powerful complementary knowledge, substantially improving commit message generation. Moreover, HierDiff-6.7B performs acceptably using only architectural information, providing a viable solution when code diffs are empty or exceed input limits.

Limitations

We summarize the limitations of this paper as follows. First, our HierCMD dataset only includes Python and Java, with no training or evaluation conducted on other programming languages. As a result, the generalizability of our proposed method to other languages remains unvalidated. Second, while we incorporate architectural information to improve commit message generation, our analysis largely stays at the data level, lacking an in-depth investigation into its unique contribution within the model. Finally, we only provide a preliminary exploration of using architectural diff alone for generation. Although its performance is acceptable, it falls considerably behind state-of-the-art models that leverage both architectural and code diffs, leaving this direction for future exploration.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant No. 62502015).

References

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, Ann Arbor, Michigan. Association for Computational Linguistics.

Eleftheria Briakou, Zhongtao Liu, Colin Cherry, and Markus Freitag. 2024. On the implications of verbose llm outputs: A case study in translation evaluation. *Preprint*, arXiv:2410.00863.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

Max Brunsfeld, Amaan Qureshi, Andrew Hlynskyi, Patrick Thomson, ObserverOfTime, Josh Vera, dundargoc, Phil Turnbull, Will Lillis, Timothy Clem, Douglas Creager, Andrew Helwer, Rob Rix, Daumantas Kavolis, Hendrik van Antwerpen, Michael Davis, Ika, Amin Ya, Tuán-Anh Nguy en, and 10 others. 2025. tree-sitter/tree-sitter: target/tree-sitter-linux- arm.gz.

Raymond P.L. Buse and Westley R. Weimer. 2010. Automatically documenting program changes. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, page 33–42, New York, NY, USA. Association for Computing Machinery.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, and 1 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *Preprint*, arXiv:2507.06261.

Di Cui, Jiaqi Guo, Ting Liu, and Qinghua Zheng. 2021. An empirical study of architectural changes in code commits. In *Proceedings of the 12th Asia-Pacific Symposium on Internetware*, Internetware '20, page 11–20, New York, NY, USA. Association for Computing Machinery.

DeepSeek-AI. 2026. Deepseek-v4: Towards highly efficient million-token context intelligence.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, and 21 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *Preprint*, arXiv:2406.11931.

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 *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 970–981.

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 *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 422–431. IEEE Press.

Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. 2023. Talk like a graph: Encoding graphs for large language models. *Preprint*, arXiv:2310.04560.

Minghua He, Yue Chen, Fangkai Yang, Pu Zhao, Wenjie Yin, Yu Kang, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. 2025a. ExeCoder: Empowering large language models with executability representation for code translation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 7099–7125, Suzhou, China. Association for Computational Linguistics.

Minghua He, Tong Jia, Chiming Duan, Huaqian Cai, Ying Li, and Gang Huang. 2024. Llmelog: An approach for anomaly detection based on llm-enriched log events. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, pages 132–143.

Minghua He, Tong Jia, Chiming Duan, Huaqian Cai, Ying Li, and Gang Huang. 2025b. Weakly-supervised log-based anomaly detection with inexact labels via multi-instance learning. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, page 2918–2930. IEEE Press.

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*, ISSTA 2023, page 792–803, New York, NY, USA. Association for Computing Machinery.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *Preprint*, arXiv:2106.09685.

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.

Yuan Huang, Qiaoyang Zheng, Xiangping Chen, Yingfei Xiong, Zhiyong Liu, and Xiaonan Luo. 2017. Mining version control system for automatically generating commit comment. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '17, page 414–423. IEEE Press.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

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.

Zhihua Jiang, Jianwei Chen, Dongning Rao, and Guanghui Ye. 2024. Leveraging context-aware prompting for commit message generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 13522–13540, Miami, Florida, USA. Association for Computational Linguistics.

Jiawei Li and Iftekhar Ahmed. 2023. Commit message matters: Investigating impact and evolution of commit message quality. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 806–817. IEEE Press.

Jiawei Li, David Faragó, Christian Petrov, and Iftekhar Ahmed. 2024. Only diff is not enough: Generating commit messages leveraging reasoning and action of large language model. *Proc. ACM Softw. Eng.*, 1(FSE).

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. 2015. Changescribe: a tool for automatically generating commit messages. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, page 709–712. IEEE Press.

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.

Shangqing Liu, Cuiyun Gao, Sen Chen, Lun Yiu Nie, and Yang Liu. 2022. Atom: Commit message generation based on abstract syntax tree and hybrid ranking. *IEEE Transactions on Software Engineering*, 48(5):1800–1817.

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*, ASE '18, page 373–384, New York, NY, USA. Association for Computing Machinery.

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 (Volume 2: Short Papers)*, pages 287–292, Vancouver, Canada. Association for Computational Linguistics.

Mockus and Votta. 2000. Identifying reasons for software changes using historic databases. In *Proceedings 2000 International Conference on Software Maintenance*, pages 120–130.

Tiago Oliveira Motta, Rodrigo Rocha Gomes e Souza, and Claudio Sant'Anna. 2018. Characterizing architectural information in commit messages: an exploratory study. In *Proceedings of the XXXII Brazilian Symposium on Software Engineering*, SBES '18, page 12–21, New York, NY, USA. Association for Computing Machinery.

Lun Yiu Nie, Cuiyun Gao, Zhicong Zhong, Wai Lam, Yang Liu, and Zenglin Xu. 2021. Coregen: Contextualized code representation learning for commit message generation. *Preprint*, arXiv:2007.06934.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, and 1 others. 2024. Gpt-4 technical report. *Preprint*, arXiv: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*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Rafiqul Rabin, Sean McGregor, and Nick Judd. 2025. Malicious and unintentional disclosure risks in large language models for code generation. *Preprint*, arXiv:2503.22760.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2023. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Yashothara Shanmugarasa, Ming Ding, Chamikara Mahawaga Arachchige, and Thierry Rakotoarivelo. 2025. Sok: The privacy paradox of large language models: Advancements, privacy risks, and mitigation. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '25, page 425–441, New York, NY, USA. Association for Computing Machinery.

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 (COMPSAC)*, volume 1, pages 103–112.

Ensheng Shi, Yanlin Wang, Wei Tao, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2022. RACE: Retrieval-augmented commit message generation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5520–5530, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Petru Soviany, Radu Tudor Ionescu, Paolo Rota, and Nicu Sebe. 2022. Curriculum learning: A survey. *Preprint*, arXiv:2101.10382.

Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 908–911, New York, NY, USA. Association for Computing Machinery.

Jinrui Sun, Tong Jia, Minghua He, and Ying Li. 2026. Varparser: Unleashing the neglected power of variables for llm-based log parsing. In *Proceedings of the ACM Web Conference 2026*, WWW '26, page 6965–6976, New York, NY, USA. Association for Computing Machinery.

Lei Tao, Shenglin Zhang, Zedong Jia, Jinrui Sun, Minghua Ma, Zhengdan Li, Yongqian Sun, Canqun Yang, Yuzhi Zhang, and Dan Pei. 2024. Giving every modality a voice in microservice failure diagnosis via multimodal adaptive optimization. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1107–1119, New York, NY, USA. Association for Computing Machinery.

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.

Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How do software engineers understand code changes? an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA. Association for Computing Machinery.

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*, ICSE '22, page 2389–2401, New York, NY, USA. Association for Computing Machinery.

Ramakrishna Vedantam, C. Lawrence Zitnick, and Devi Parikh. 2015. Cider: Consensus-based image description evaluation. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4566–4575.

Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. 2021. Context-aware retrieval-based deep commit message generation. *ACM Trans. Softw. Eng. Methodol.*, 30(4).

Xindi Wang, Mahsa Salmani, Parsa Omidi, Xiangyu Ren, Mehdi Rezagholizadeh, and Armaghan Eshaghi. 2024. Beyond the limits: a survey of techniques to extend the context length in large language models. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, IJCAI '24.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: empowering code generation with oss-instruct. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

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. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, page 553–565. IEEE Press.

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 Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 3975–3981. International Joint Conferences on Artificial Intelligence Organization.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2024. WaveCoder: Widespread and versatile enhancement for code large language models by instruction tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5140–5153, Bangkok, Thailand. Association for Computational Linguistics.

Qunhong Zeng, Yuxia Zhang, Zexiong Ma, Bo Jiang, Ningyuan Sun, Klaas-Jan Stol, Xingyu Mou, and Hui Liu. 2025. Evaluating generated commit messages with large language models. *Preprint*, arXiv:2507.10906.

A Threats to Validity

We have made deliberate efforts to mitigate potential threats that could undermine the validity of this study. The identified threats are discussed as follows.

Privacy Concern. Since CMG inherently requires processing sensitive or proprietary code, the reliance on external LLM APIs poses inherent privacy risks, especially when handling proprietary or sensitive source code in real-world development settings. To eliminate such risks, our approach is built entirely around a local LLM that operates within a self-contained environment without transmitting any code data to external servers.

Reproducibility and Settings. Our experimental results may be influenced by design choices such as prompt templates, decoding strategies, and hyperparameter settings. To reduce this threat, we

apply identical generation settings and prompt templates across all baseline methods. Furthermore, the temperature is set to 0 for all experiments to eliminate stochasticity in the output and ensure reproducibility. Finally, we provide the reproduction scripts for all open-source and closed-source models in our source code.

B Experiments Supplement

B.1 Metrics

BLEU measures the n-gram precision between the generated text and the reference text. ROUGE-L is a recall-oriented metric that evaluates the longest common subsequence between the generated text and the reference text. METEOR assesses the alignment between generated and reference texts by considering exact matches, stemmed matches, and synonym matches. CIDEr treats each sentence as a document and computes the cosine similarity between the TF-IDF vectors of n-grams, thereby measuring the similarity between the generated text and the reference text. In addition to these automatic metrics, we further conduct human evaluation to provide a more comprehensive assessment of commit message quality. For human evaluation, experienced developers are invited to evaluate the generated messages, allowing us to better capture the semantic correctness and practical usefulness that may not be fully reflected by automatic metrics alone. For LLM-based evaluation, we employed zero-shot in-context learning on deepseek-v4-flash (DeepSeek-AI, 2026), assessing the *What* and *Why* dimensions using a 5-point Likert scale. The prompt provided to the LLM is: "You are an experienced software engineer. Your task is to evaluate the commit message based on the corresponding git commit diff, focusing on two dimensions: <dimensional definitions> + <scoring criteria> + <commit to be evaluated>."

B.2 Implementation Details

We adopt DeepSeek-Coder-6.7B-Instruct as the base model for fine-tuning. The model is trained using supervised fine-tuning with a maximum sequence length of 2048 tokens. The effective batch size is set to 16, achieved via a per-device batch size of 2 and gradient accumulation over 4 steps. We train the model for multiple epochs using the AdamW optimizer with an initial learning rate of 2e-4, together with a cosine learning rate scheduler. Mixed-precision training with FP16 is applied to improve training efficiency and reduce memory consumption. During training, model checkpoints are saved at the end of each epoch, with at most two checkpoints retained. Model selection is based on the validation loss, and the best-performing checkpoint is loaded for evaluation.

C Prompt Designs

Staged Diff Learning consists of three stages, each targeting a distinct granularity of change information and guided by stage-specific prompts (summarized in Table 7).

D Datasets Supplement

D.1 Dataset Format

Figure 7 illustrates a case from HierCMD to demonstrate the specific data format. Specifically, the code diff is encapsulated within the *changed files* field, while the *arch_diff_graph* field captures the architectural change information.

D.2 Dataset Comparison

Several datasets have been developed for the commit message generation task (Jiang et al., 2017; Liu et al., 2018; Xu et al., 2019; Liu et al., 2019; Loyola et al., 2017; Liu et al., 2022; Tao et al., 2021; Jiang et al., 2024; Wu et al., 2025). A brief overview of these datasets is provided below:

- **NNGen** (Liu et al., 2018): A cleaned-up version of CmtGen, NNGen removes trivial or bot-generated commit messages to improve data quality.
- **CoDiSum** (Xu et al., 2019): Based on the top 1,000 Java repositories, CoDiSum employs a file-level programming language filter to select relevant commits.
- **PtrGNCMsg** (Liu et al., 2019): This dataset follows the same preprocessing pipeline as CmtGen but expands the repository coverage to the top 2,000 Java projects.
- **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., 2022): Collected from the 56 most-starred Java projects, ATOM filters out commits with noisy content or those that do not modify source code. It provides not only the

Repository: google/guava

{
'commit_hash': '8ea070f483c623678e1c642ae0b9b4c70c5222c2',
'commit_msg': 'Disable two assertions for j2kt that are only passing by accident',
'date': '2025-02-10 10:30:38-08:00',
'changed_files': [{'filename': 'ContiguousSetTest.java', 'old_path': 'android/guava-tests/test/com/google/common/collect/ContiguousSetTest.java', 'new_path': 'android/guava-tests/test/com/google/common/collect/ContiguousSetTest.java', 'added_lines': 12, 'deleted_lines': 0, 'diff': '.....'}, ...],
'arch_diff_graph': {'added_nodes': [], 'removed_nodes': [], 'modified_nodes': [], 'added_edges': [{'src': 'file::android/guava-tests/test/com/google/common/collect/ContiguousSetTest.java', 'dst': 'class::android/guava-testlib/src/com/google/common/collect/testing/google/SetGenerators.java::ContiguousSetDescendingGenerator', 'type': 'import'}, {'src': 'file::guava-tests/test/com/google/common/collect/ContiguousSetTest.java', 'dst': 'class::android/guava-testlib/src/com/google/common/collect/testing/google/SetGenerators.java::ContiguousSetDescendingGenerator', 'type': 'import'}], 'removed_edges': [], 'modified_files': ['guava-tests/test/com/google/common/collect/ContiguousSetTest.java', 'android/guava-tests/test/com/google/common/collect/ContiguousSetTest.java']}}

Figure 6: An example of the data format in the HierCMD.

Repository: oracle/graal

{
'commit_hash': '3485654cbe00e3094970ef08b9c5d5e6098df780',
'commit_msg': 'Only collect init-time and eval-time if the metric is time.',
'date': '2022-06-20T08:15:12Z',
'diff': "diff --git a/vm/mx.vm/mx_vm_benchmark.py b/vm/mx.vm/mx_vm_benchmark.py\nindex 7af9ad266f33..3d5b65b5e03a 100644\n--- a/vm/mx.vm/mx_vm_benchmark.py\n+++ b/vm/mx.vm/mx_vm_benchmark.py\n@@ -1245,28 +1245,6 @@ def polybenchmark_rules(benchmark, metric_name, mode):\n ...",
'arch_diff_graph': {"added_nodes":[],"removed_nodes":[],"modified_nodes":[],"added_edges":[],"removed_edges":[{"src":"func::vm/mx.vm/mx_vm.py::mx_post_parse_cmd_line","dst":"func::vm/mx.vm/mx_vm_benchmark.py::register_graalvm_vms","type":"invoke"},{"src":"func::vm/mx.vm/mx_vm_gate.py::_jdk_has_ForceTranslateFailure_jvmci_option","dst":"func::vm/mx.vm/mx_vm_benchmark.py::run","type":"invoke"},{"src":"func::vm/mx.vm/mx_vm_gate.py::_test_libgraal_basic","dst":"func::vm/mx.vm/mx_vm_benchmark.py::run","type":"invoke"},{"src":"func::vm/mx.vm/mx_vm_gate.py::gate_svm_sl_tck","dst":"func::vm/mx.vm/mx_vm_benchmark.py::run","type":"invoke"}],"modified_files":["vm/mx.vm/mx_vm_benchmark.py"]}}

Figure 7: An example of the data format in the MCMD-New-TestX.

Input	Prompt
Architectural Diff	The repository diff contains both code changes and architectural changes. Only architectural changes are provided below. File-level architectural diffs are enclosed between `<File Diff>` and `</File Diff>` tags. Output a concise commit message with no additional text.
Code Diff	The repository diff contains both code changes and architectural changes. Code diffs are enclosed between `<Code Diff>` and `</Code Diff>` tags, with module-level architectural diffs provided as inline comments within the code diffs. Output a concise commit message with no additional text.
Hierarchical Diff	The repository diff contains both code changes and architectural changes. Architectural diffs include file-level and module-level diffs. File-level architectural diffs are enclosed between `<File Diff>` and `</File Diff>` tags. Code diffs are enclosed between `<Code Diff>` and `</Code Diff>` tags, with module-level architectural diffs provided as inline comments within the code diffs. Output a concise commit message with no additional text.

Table 7: Prompts used for different input types.

original commit messages but also the specific functions affected by each commit.

- **MCMD** (Tao et al., 2021): A large dataset without filtering, MCMD includes programs in five programming languages drawn from the top 500 GitHub repositories.
- **CODEC** (Jiang et al., 2024): This dataset gathers Java repositories on GitHub with at least 1,000 stars and includes context lines to provide richer information.
- **MCMD-New** (Wu et al., 2025): Containing 229,492 commits from 367 repositories that also appear in MCMD, this dataset focuses on commits collected after January 1, 2022.