Language-specific fine-tuning of transformer models for commit message generation in version control systems*

Sergiy Pogorilyy[1,†] and Bohdan Semonov[1,*,†]

[1] Taras Shevchenko National University of Kyiv, 60 Volodymyrska Street, Kyiv, 01033, Ukraine

Abstract

Automatic generation of commit messages from source code changes is an important task in software engineering aimed at improving documentation and maintenance. This paper analyzes transformer-based models – T5, FLAN-T5, CodeT5, CodeT5+, and BART – for commit message generation from code diffs. Several fine-tuning strategies are explored, including instruction tuning, multi-task learning, and language-specific fine-tuning for version control systems. Experiments are conducted on source code written in C/C++, Ruby, Python, Java, JavaScript (JS), and PHP, enabling an analysis of the impact of programming language characteristics on generation quality. The models are evaluated using BLEU, ROUGE-1, ROUGE-2, and ROUGE-L, as well as semantic metrics BERTScore and BLEURT. The results show that instruction-aware and language-specific fine-tuning improves performance compared to baseline models, while also highlighting the limitations of lexical metrics for semantic quality assessment.

Keywords

BART, BERTScore, BLEU, CodeT5, T5, commit message generation, semantic evaluation, source code, transformer models

1. Introduction

Version control systems emerged several decades ago and have since undergone significant evolution: from centralized to distributed architectures, and from storing complete file copies to recording only the differences between versions. One of the key elements of such systems is the commit message, which serves as technical documentation of project modifications and plays an essential role in the software development process.

Unlike traditional technical documentation, commit messages lack clearly defined standards or established writing conventions. This gives rise to a number of challenges, including the unification of style, structure, and completeness of the provided information. The absence of standardized guidelines does not ensure that change messages within a single project will contain consistent, comprehensive, and properly formulated details regarding the nature of the changes, their rationale, implications, and implementation specifics [1].

As a result, a wide range of informal practices has emerged. Developers have gradually established their own guidelines for writing commit messages, which often depend on the specifics of the domain, the type of software, and the technologies and tools in use. In some cases, these practices may even be mutually exclusive. At the same time, there exists a set of general principles whose adherence ensures the creation of high-quality change documentation, whereas their violation indicates low informativeness or inadequate message structure [2].

Under these conditions, there is an urgent need for the development of automated tools for generating meaningful and standardized commit messages. With the advent of transformer-based models such as BART, T5, and CodeT5, it has become possible to effectively apply natural language processing (NLP) methods to tasks related to source code. These models have already demonstrated

*ITS-2025: Information Technologies and Security, December 11, 2025, Kyiv, Ukraine

* Corresponding author.

† These authors contributed equally.

sdp@knu.ua (S. Pogorilyy); bohdan.semonov@gmail.com (B. Semonov)

0000-0002-6497-5056 (S. Pogorilyy); 0009-0001-3692-9415 (B. Semonov)

© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-4337/paper21.pdf

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-26

strong performance in code summarization, translation between programming languages, and documentation generation [3].

Rules for formulating descriptions of code changes

1. Structural Requirements for the Message [4]:
    a. the description must include a header and, if necessary, a body. These elements should be separated by a blank line;
    b. the header should be concise, with a length not exceeding 50 – 70 characters;
    c. verbs in the header should be used in the imperative mood.
2. Content Requirements for the Message [4]:
    a. the reasons for the changes must be specified;
    b. the expected impact of the changes should be described;
    c. in the case of a bug fix, the specific issue that was resolved should be indicated;
    d. the content of the message should be presented in such a way that even a specialist unfamiliar with the details of the problem or the code structure can understand the essence of the changes and their impact on the software project.

Examples of descriptions are provided in Table 1.

Table 1
Examples of commit messages compliant with Rule 2.d

No.	Commit message example	Message purpose
1	Add validation for empty fields in user registration form	To prevent situations where the user leaves the form incomplete, which may lead to errors in the database.
2	Fix incorrect total price calculation in shopping cart	To ensure the correctness of financial operations and increase user trust in the system.
3	Add password reset functionality via email	To improve system usability and reduce the number of support requests.
4	Fix time display for different time zones	To ensure data accuracy for users across different regions.
5	Add success notification after saving user settings	To provide the user with feedback after performing an action.
6	Update password hashing algorithm to enhance security compliance	To enhance security in accordance with the requirements.
7	Fix race condition in data loader to prevent intermittent test failures	To eliminate the cause of unstable test execution.

2. Problem formulation

In the context of modern software development, particularly when using version control systems, commit messages constitute an important element of technical documentation. They are expected to provide a concise yet informative description of changes in the codebase, their rationale, and their impact. At the same time, the absence of unified standards and guidelines for writing such messages results in substantial variability in both structure and content, which complicates the analysis of change histories, change tracking, and code reuse.

Against the backdrop of rapid advancements in natural language processing (NLP), transformer architectures, particularly BART, T5, and CodeT5, have demonstrated high effectiveness in tasks such as text summarization, automatic code translation, documentation generation, and other software-oriented applications. This provides a foundation for leveraging these models for the

automated generation of commit messages based on textual representations of changes in the codebase.

Accordingly, the main objective of this study is to develop and experimentally evaluate an approach to automatic generation of commit messages from code changes using transformer-based models with language-specific fine-tuning. The study is conducted on source code written in C/C++, Ruby, Python, Java, JavaScript (JS), and PHP, enabling an analysis of the impact of programming languages on the effectiveness of commit message generation. To achieve this objective, the following tasks are addressed:

1. selecting an appropriate transformer architecture and performing fine-tuning, including language-specific fine-tuning, on a corpus of "code changes–commit message" pairs;
2. preprocessing textual diffs extracted from open-source repositories and constructing training and test datasets separately for each programming language;
3. defining evaluation metrics to assess the quality of the generated commit messages in comparison with human-written references;
4. analyzing the advantages and limitations of the proposed approach in terms of both the level of automation and the semantic quality of the obtained results.

Problem Formalization

Let:

1. $x \in X$ denote a textual representation of code changes (e.g., a *git diff*);
1. $y \in Y$ denote a commit message describing the changes;
2. $D = \left\{ \left(x_i, y_i \right) \right\}_{i=1}^{N}$ denote a corpus of examples collected from repositories, where each pair consists of code changes and the corresponding commit message;
3. $f_\theta : X \to Y$ denote a parameterized transformer-based model with parameters θ, which approximates the conditional probability $P\left(y \mid x ; \theta \right)$.

The objective is to find the optimal parameters θ^{opt} that minimize the loss function L on the validation dataset:

$$\theta^{opt} = arg\min_{\theta} \sum_{(x_i, y_i) \in D} L\left(f_\theta\left(x_i \right), y_i \right), \quad (1)$$

where L denotes the cross-entropy loss function or a similarity-based metric between the generated and the reference (ground-truth) commit message.

After training, for a new example $x^{test} \in X$, the model generates a commit message:

$$y^{test} = f_{\theta^{opt}}\left(x^{test} \right). \quad (2)$$

3. Collection and preparation of training corpora for commit messages

To collect training data, we employed GitHub, one of the largest web services for hosting IT projects and collaborative software development, which is based on the well-known version control system Git. This service provides a dedicated application programming interface, referred to as the GitHub REST API [5]. Since the study is focused on analyzing the content of commit messages describing code modifications (differences, or "diffs") in version control systems, the following interfaces were utilized:

1. retrieving a list of repositories (projects);
2. retrieving commit messages for each repository;
3. retrieving the code modifications associated with each commit message.

The aforementioned services are implemented as HTTP GET requests, which must construct and transmit the appropriate GET parameters and HTTP headers to the server. In turn, the responses from these services are returned in the JSON text format of structured data [6].

Table 2
Features of the GitHub REST API services

Service Name	URL	GET parameters	Research-relevant fields
Retrieving the list of repositories	https://api.github.com/repositories	*since* – a parameter whose value must be an integer corresponding to the repository identifier.	*id* – repository identifier; *full_name* – repository name; *description* – detailed description of the repository.
Retrieving commit messages for each repository	https://api.github.com/repos/[full_name]/commits?per_page=100, where *full_name* refers to the repository name from the previous row of the table	*per_page* – the number of commit messages returned in a single request; *sha* – the character string (hash) of the commit message from which the search should start.	*message* – the commit message, located within the JSON object "commit".
Retrieving code changes for each commit message	https://api.github.com/repos/[full_name]/commits/[sha], where *full_name* refers to the repository name from the first row of the table; *sha* – the hash of the commit message from the second row of the table	*None*	*filename* – the name of the modified file; *diff* – the portion of the file that was changed, compared to the previous version.

The HTTP requests listed in Table 2 share identical sets of HTTP headers, namely:

1. 'Accept': 'application/vnd.github+json' – the response type returned by the application programming interface (GitHub REST API);
2. 'Authorization': 'Bearer [token]' – API user authorization, where token refers to a special character string that can be generated in the user settings of the GitHub web service;
3. 'X-GitHub-Api-Version': '2022-11-28' – the version of the GitHub REST API.

After data retrieval, the dataset underwent preliminary filtering, as it contained commit messages varying in stylistic presentation, length, and level of detail regarding the purpose and content of modifications within the project. This indicates that a portion of the messages did not comply with the structural and content requirements formulated in the first section. All established rules can be implemented programmatically, except for Rule 2.d, for which traditional formal methods are insufficient. Due to the lack of clearly structured criteria for identifying the substantive aspects of changes and their motivations, the implementation of this rule requires the use of machine learning techniques capable of modeling a semantic "understanding" of the context of commit messages.

To implement the aforementioned approach, a method for filtering commit message descriptions was developed based on MLP and RNN models. It was determined that this filter performs a binary classification task, which can be described by a surjective, but not injective, mapping. [7]:

$$f: X \to Y, \quad (3)$$

where X is the set of commit message descriptions, each of which is represented by an m-dimensional word vector: $\vec{x} = \{x_1, x_2, \ldots, x_m\}, \vec{x} \in X$;

Y – the set of possible responses (labels) represented as M-dimensional vectors: $\vec{y} = \{y_1, \ldots, y_M\}, \vec{y} \in Y$.

For model training, manually pre-labeled data in accordance with Rule 2.d were used, resulting in a collection of 8,000 messages.

Prior to inputting the messages into the models, the filter performed a series of preprocessing steps [8]:

1. text normalization, including converting all letters to lowercase;
2. tokenization of messages in the training set;
3. vectorization of messages, with the output token sequence length fixed to ensure a consistent tensor shape at the neural network input, and with vocabulary size limited to discard infrequently used tokens for computational efficiency.

As a result, seven filtering models were constructed: two based on multilayer perceptrons (MLP) and five based on recurrent neural networks (RNN). The architecture of each model included:

1. two weighted layers:
    a. an input embedding layer;
    b. an output fully connected (Dense) layer with the number of neurons corresponding to the number of classes (M = 2) [9];
2. an intermediate layer without weights (Global Average Pooling) [9];
3. the AdamW optimizer [10], which is a variant of the Adam algorithm incorporating weight decay regularization. This choice demonstrated faster training and improved generalization;
4. a Dropout layer with a rate of 0.3, which prevents overfitting by randomly deactivating neurons during each training iteration [11].

To implement the aforementioned models, the Python programming language (version 3.12.11) was used, along with the TensorFlow library (2.19.0).

The model performance was evaluated using the Accuracy and F1-score metrics [12], with the latter representing the harmonic mean of precision and recall. Based on the results (see Table 3), the best candidate model was selected.

Table 3

Comparison of the performance of constructed models for implementing the commit message filter

No.	Model	Metric	Value
1	MLP without regularization techniques	Accuracy	0.786
		F1-score	0.622
2	MLP with regularization techniques	Accuracy	0.816
		F1-score	0.648
3	LSTM (48 neurons)	Accuracy	0.801
		F1-score	0.445
4	BiLSTM (1 layer with 24 neurons)	Accuracy	0.796
		F1-score	0.664
5	BiLSTM (2 layers with 12 neurons each)	Accuracy	0.794
		F1-score	**0.677**
6	BiGRU (1 layer with 24 neurons)	Accuracy	**0.808**
		F1-score	**0.651**
7	BiGRU (2 layers with 12 neurons each)	Accuracy	0.791
		F1-score	0.635

From the results presented in the table, it is evident that the best performance was achieved by an RNN containing a single bidirectional GRU layer with 24 neurons for text feature extraction: the F1-score was 65.1%, and the recall was 80.8%. Consequently, this model was selected as the filter for commit messages, with its architecture shown in Figure 1 [6].

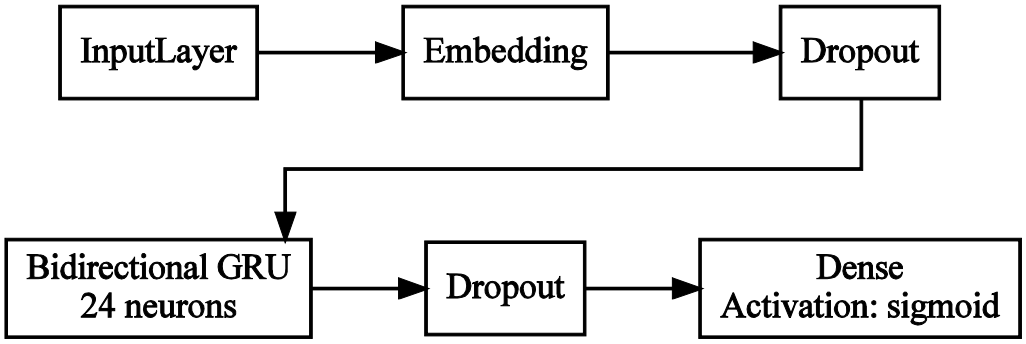


Figure 1: Architecture of the model with a single recurrent BiGRU layer and regularization techniques.

Using the filter, commit messages complying with Rule 2.d were selected. Based on these messages and the corresponding code changes, training and validation corpora were constructed for several popular programming languages: C/C++, Ruby, Python, Java, JavaScript (JS), and PHP. A different number of "change–message" pairs was used for each language. The distribution of data for training and evaluation for each language is presented in Table 4. It is noteworthy that a single "change-message" pair may include between one and five files, depending on the scale of the modifications.

Table 4
Distribution of "change-message" pairs for training and validation

No.	Programming language	Number of training pairs	Number of validation pairs
1	C/C++	149 357	29 871
2	Ruby	28 707	5 741
3	Python	34 473	6 894
4	Java	27 339	5 467
5	JavaScript	16 876	3 375
6	PHP	14 752	2 950

4. Method for generating commit messages in source code version control systems

To address the task of generating commit messages, a range of modern transformer-based models was applied, including: Salesforce/codet5-small, Salesforce/codet5-base, Salesforce/codet5-large, Salesforce/codet5p-220m, Salesforce/codet5p-770m, t5-base, t5-large, google/flan-t5-base, and facebook/bart-large. The main characteristics of these models are presented in Table 5.

The models were selected based on their architecture and the degree of adaptation to code-related tasks (e.g., CodeT5) or to general-purpose text summarization and generation tasks (e.g., FLAN-T5, BART). The architectures are shown in Figure 2.

All models were fine-tuned on a precompiled corpus of "change-message" pairs for several programming languages (C/C++, Ruby, Python, Java, JavaScript (JS), PHP). During fine-tuning, the respective models' native tokenizers (AutoTokenizer from the HuggingFace Transformers library) were used [17].

The choice of the Hugging Face Transformers library (a comparison with other libraries is presented in Table 6) was motivated by the availability of models pre-trained on source code, as well as the ability to deploy them within the Google Colab environment.

Table 5
Main parameters of pretrained models

Model Name	Architecture	Parameters (M)	Vocabulary (K)	Organization	Key Features
codet5-small	T5	60	32	Salesforce	Pre-trained on source code, suitable for rapid prototyping [13].
codet5-base	T5	220	32	Salesforce	Baseline variant of CodeT5, supports multiple tasks: generation, summarization.
codet5-large	T5	770	32	Salesforce	More powerful version, but with higher resource requirements.
codet5p-220m	T5 (CodeT5+)	220	32	Salesforce	Improved generation quality through instruction tuning [14].
codet5p-770m	T5 (CodeT5+)	770	32	Salesforce	Enhanced quality for source code generation.
t5-base	T5	220	32	Google	General-purpose model, not specifically trained on source code.
t5-large	T5	770	32	Google	Larger variant of t5-base, better performance on language tasks.
flan-t5-base	T5 (Flan)	220	32	Google	Instruction-tuned, improved generalization [15].
bart-large	BART	400	50	Meta (Facebook)	Encoder-decoder architecture, specialized in text rephrasing [16].

Training was conducted with fixed hyperparameters:

1. batch size: 4 or 8;
2. number of epochs: 50 (with early stopping, patience set to 5);
3. optimizer – AdamW [10] with an initial learning rate of 5×10^{-5};
4. weight decay: 0.01;
5. use of a linear learning rate scheduler.

The training environment was chosen as Google Colaboratory [20], as it provides users (researchers, developers) with cloud-based hardware resources for GPGPU and TPU technologies, and allows the use of all aforementioned machine learning libraries.

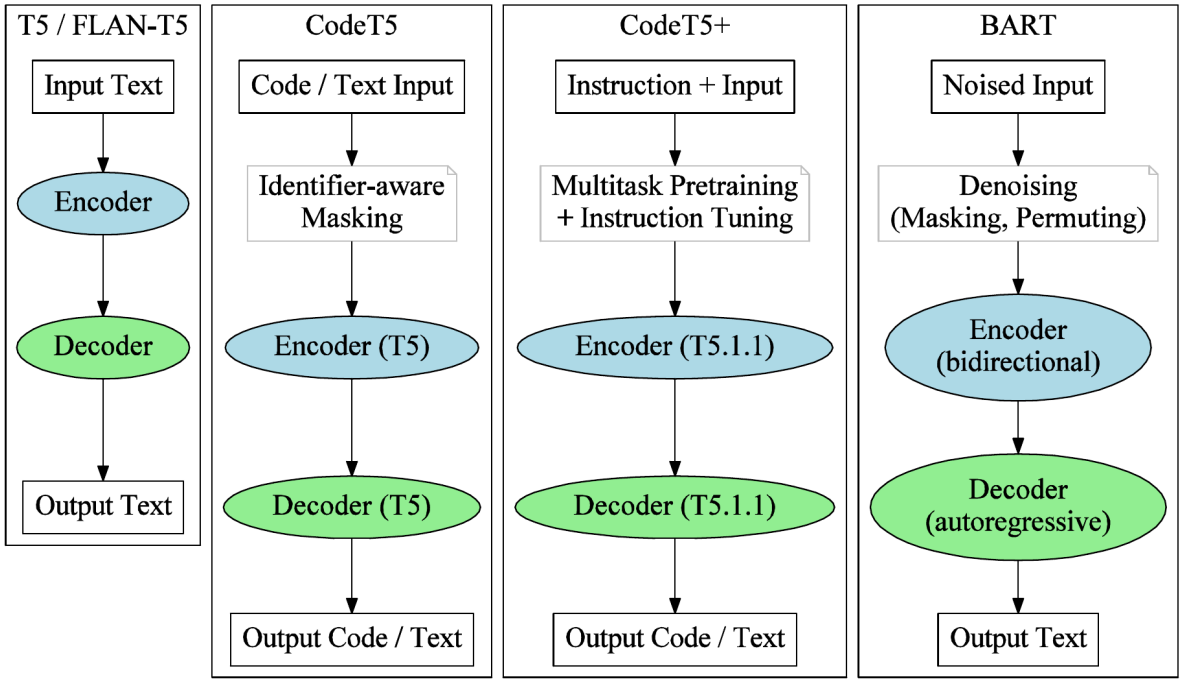


Figure 2: Schematic representation of the architectures of T5, CodeT5, CodeT5+, FLAN-T5, and BART models.

Since some models employed in this work contain over 100 million parameters, substantial computational resources (both processing time and, especially, memory) are required for their training. Such resources are provided by NVIDIA T4, L4, and A100 GPUs, which are among the most powerful graphics processors designed for deep learning, high-performance computing (HPC), and cloud-based computations. These GPUs are commonly available in Google Colab Pro/Pro+, AWS, Azure, NVIDIA DGX systems, and supercomputers. Key specifications are presented in Table 7.

Table 6
Comparative overview of libraries and frameworks for commit message generation

Library / framework	Support for code models	Commit Message Generation	Google Colab Compatibility	Primary framework
Hugging Face [17]	Yes (CodeT5, PLBART, etc.)	Yes	Yes	PyTorch / TensorFlow
OpenNMT-py [1]	Partially (basic Seq2Seq)	Possible manually	Yes	PyTorch
Fairseq (Meta) [18]	Limited (BART, T5 can be adapted)	Yes (with manual tuning)	Yes	PyTorch
T5X (Google)	Yes (CodeT5+)	Requires verification	No (does not fully support Colab/TPU)	JAX / Flax
ESPnet / ESPnet2 [19]	No	No	No	PyTorch
OpenAI API (Codex, GPT-4) [3]	Yes (via API)	No	Yes	API
CodeReviewer (NCSU)	Yes (specialized model)	Partially	No	PyTorch + GitHub API

Table 7
Comparative characteristics of graphics processing units

GPU	Architecture	Number of CUDA cores	Number of tensor cores	Memory
NVIDIA T4	Turing (TU104)	2 560	320	16 GB GDDR6
NVIDIA L4	Ada Lovelace (AD104)	7 424	232	24 GB GDDR6
NVIDIA A100	Ampere (GA100)	6 912	432	40 GB HBM2e

5. Evaluation of the method for generating commit messages

In tasks of automatic text quality evaluation, such as machine translation, summarization, and text generation, various metrics are employed to measure the similarity between generated text and reference text. Among the most commonly used metrics are BLEU and ROUGE, which assess quality based on the number of shared units (n-grams) between the predicted and reference texts. Each of these metrics has specific characteristics and applications, which are described below.

BLEU (Bilingual Evaluation Understudy) is one of the primary metrics for assessing the quality of machine translation. It is based on comparing n-grams of the generated text with those of the reference text. The main evaluation steps include [21]:

1. determining the number of exact n-gram matches (typically for n = 1, 2, 3, 4);
2. applying a brevity penalty, which reduces the score in cases where the generated translation is significantly shorter than the reference.

Mathematically, BLEU measures the similarity of the generated text based on n-gram precision, giving greater weight to higher-order n-grams. An important aspect is that BLEU focuses on precision and does not consider recall, which can lead to biased results in cases where not only exact matches but also omitted elements are significant.

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a set of metrics used to assess the quality of text generation, particularly in tasks such as summarization and text generation. A distinguishing feature of ROUGE is its focus on recall, as well as the computation of the number of overlapping n-grams between the generated and reference texts [22]. ROUGE has several variants, including:

1. ROUGE-1 – measures the overlap of unigrams (single words) between the generated and reference texts. It provides an indication of the model's overall ability to capture content at the level of individual units;
2. ROUGE-2 – measures the overlap of bigrams (two-word sequences) between texts. ROUGE-2 emphasizes contextual relationships and the coherence between words;
3. ROUGE-L – calculates the length of the Longest Common Subsequence (LCS) between two texts. It reflects the model's ability to preserve word order and text structure.

BERTScore is a modern metric that leverages contextual word embeddings obtained from a pre-trained BERT transformer model. It evaluates the semantic similarity between tokens in the generated and reference texts by computing the cosine similarity between their embeddings. Consequently, BERTScore captures not only surface-level word matches but also semantic correspondence, making it particularly useful for assessing text generation quality when preserving meaning is critical [23].

BLEURT (Bilingual Evaluation Understudy with Representations from Transformers) is a metric trained on human annotations that combines the strengths of traditional automatic approaches with neural models. BLEURT is based on pre-trained language models (e.g., BERT) and further fine-tuned on corpora annotated with human quality ratings. This enables the metric to capture more subtle aspects of text quality, including grammar, style, fluency, and semantic equivalence.

BLEURT exhibits a high correlation with human judgments in machine translation and text generation tasks [24].

BLEU is frequently employed for evaluating machine translation, where precision relative to reference translations is critical. However, it may exhibit limitations in tasks requiring diverse text generation, as the metric does not account for variability and can be highly sensitive to exact matches, potentially yielding low scores in such contexts.

ROUGE, in turn, is better suited for tasks where the model's ability to capture context and content is crucial. For instance, in automatic summarization tasks, it is important to evaluate how effectively the model conveys the main ideas (semantic content) of the text, rather than merely exact word matches.

BERTScore and BLEURT are more advanced metrics that account for the semantic level of text similarity. They provide a more accurate reflection of text quality in tasks where variability in phrasing is natural and desirable.

All of these metrics have certain limitations. For instance, BLEU may underestimate models that generate text containing novel or variable expressions. Conversely, ROUGE may overestimate text quality if the model includes many extraneous words that coincide with the reference. It is also important to note that traditional metrics (BLEU, ROUGE) do not account for semantic adequacy, whereas modern metrics (BERTScore, BLEURT) allow for a more accurate reflection of text quality in alignment with human judgments.

Consequently, the results of the model evaluations are presented in Table 8.

Table 8
Comparison of the performance of developed models for commit message generation

Model	Programming language	BLEU	ROUGE-1	ROUGE-2	ROUGE-L	BERTScore	BLEURT
codet5-small	C/C++	3.54	24.57	12.26	22.87	0.88	0.34
	Ruby	4.39	23.89	9.51	21.61	0.86	0.33
	Python	3.96	24.57	11.33	22.66	0.87	0.34
	Java	3.74	24.91	14.12	23.4	0.87	0.34
	JS	**12.5**	**34.39**	**23.67**	**32.71**	**0.9**	**0.41**
	PHP	3.78	19.45	6.72	18.27	0.86	0.28
codet5-base	C/C++	3.78	23.64	11.99	22.03	0.87	0.34
	Ruby	5.74	25.39	11.14	22.9	0.86	0.34
	Python	5.9	27.46	14.74	25.31	0.87	0.36
	Java	5.03	27.56	17.56	25.94	0.88	0.36
	JS	**13.99**	**36.15**	**25.69**	**34.37**	**0.9**	**0.41**
	PHP	4.13	24.71	8.73	22.8	0.87	0.31
codet5-large	C/C++	4.75	27.62	16.24	25.89	0.88	0.36
	Ruby	6.29	25.92	11.8	23.51	0.86	0.35
	Python	5.93	28.38	15.49	26.33	0.87	0.37
	Java	5.58	28.46	18.43	26.94	0.88	0.36
	JS	**14.04**	**36.01**	**26.04**	**34.54**	**0.89**	**0.4**
	PHP	4.22	24.46	8.8	22.56	0.87	0.32
codet5p-220m	C/C++	3.92	24.29	12.24	22.5	0.8	0.33
	Ruby	5.71	24.95	11.24	22.7	0.86	0.35
	Python	5.98	27.92	15.24	25.83	0.87	0.35
	Java	5.14	27.4	17.22	25.8	0.88	0.35
	JS	**13.52**	**35.1**	**24.79**	**33.43**	**0.9**	**0.4**
	PHP	4.11	23.93	8.93	22.12	0.87	0.31
codet5p-770m	C/C++	3.92	24.34	12.36	22.54	0.88	0.35
	Ruby	6.16	26.08	11.79	23.67	0.86	0.35
	Python	6.22	27.87	15.04	25.8	0.87	0.36
	Java	5.32	28.17	18.03	26.59	0.88	0.36
	JS	**14.27**	**35.64**	**25.53**	**34.01**	**0.9**	**0.4**
	PHP	4.3	24.64	9.09	23	0.87	0.32
t5-base	C/C++	0.82	9.51	2.37	8.83	0.83	0.25

Model	Programming language	BLEU	ROUGE-1	ROUGE-2	ROUGE-L	BERTScore	BLEURT
						0.82	
	Ruby	2.59	11.02	4	10.49	0.75	0.24
	Python	0	0	0	0	0.79	0.02
	Java	0	0	0	0	0.82	0.04
	JS	1.02	9.72	2.66	8.89	0.84	0.23
	PHP	0.99	14.55	3.21	13.45		0.23
t5-large	C/C++	0.3	3.9	0.67	3.67	0.78	0.16
	Ruby	0.64	6.06	0.99	5.6	0.79	0.20
	Python	0.35	4.43	0.52	4.09	0.78	0.17
	Java	0.11	2.92	0.22	2.65	0.76	0.17
	JS	0.22	3.57	0.38	3.33	0.77	0.15
	PHP	0.36	4.03	0.53	3.74	0.78	0.15
flan-t5-	C/C++	0.42	5.07	0.93	4.74	0.8	0.17
base	Ruby	1.27	7.62	1.52	6.86	0.8	0.18
	Python	0.64	6.05	0.99	5.47	0.8	0.18
	Java	0.31	4.17	0.47	3.61	0.78	0.18
	JS	0.37	5.13	0.68	4.72	0.79	0.16
	PHP	0.28	4.07	0.56	3.75	0.77	0.14
bart-large	C/C++	2.58	21.52	10.3	19.87	0.87	0.32
	Ruby	4.82	22.36	9.12	20.09	0.86	0.32
	Python	4.59	25	12.83	23.07	0.87	0.34
	Java	4.36	24.79	15.63	23.39	0.87	0.32
	JS	**11.62**	**32.57**	**23.4**	**31.21**	**0.89**	**0.38**
	PHP	2.76	21.8	6.21	19.9	0.87	0.31

6. Conclusion

The study evaluated transformer-based models – T5, CodeT5, CodeT5+, FLAN-T5, and BART – for automatic commit message generation. Models were fine-tuned on domain-specific data and assessed using both n-gram metrics (BLEU, ROUGE) and semantic metrics (BERTScore, BLEURT).

Results showed that instruction-tuned models, particularly CodeT5+, generate more coherent and contextually relevant messages. Among all models, codet5p-770m (fine-tuned on JavaScript) achieved the highest performance: BLEU = 14.27%, ROUGE-1 = 35.64%, ROUGE-2 = 25.53%, ROUGE-L = 34.01%, BERTScore = 0.9, and BLEURT = 0.4.

The obtained results confirm the effectiveness of transformer-based models for the task of commit message generation. They also emphasize the importance of adapting training strategies and selecting appropriate evaluation metrics to ensure an objective assessment of model performance. Overall, the study demonstrates that instruction-tuned transformer architectures, combined with comprehensive evaluation approaches, provide a reliable and effective solution for generating contextually accurate commit messages.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] Jiang S., Armaly A., McMillan C., Automatically generating commit messages from diffs using neural machine translation, in: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, 2017.

[2] Buse R., Weimer W., Automatically documenting program changes, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 2010.

[3] P. Xue, L. Wu, Z. Yu, Z. Jin, Z. Yang, X. Li, Z. Yang, Y. Tan, Automated Commit Message Generation With Large Language Models: An Empirical Study and Beyond, IEEE Trans. Softw. Eng. 50.12 (2024) 3208–3224. doi:10.1109/TSE.2024.3478317.
[4] How to Write a Git Commit Message, 2014. URL: https://cbea.ms/git-commit/#seven-rules.
[5] GitHub Docs, GitHub REST API. URL: https://docs.github.com/en/rest?apiVersion=2022-11-28.
[6] B. Semonov, S. Pogorilyy, Research of the application of GPGPU and TPU technologies for ensuring comment quality in version control systems, PROBL. PROGRAM. No. 1 (2025) 24–37. doi:10.15407/pp2025.01.024.
[7] S. Pogorilyy, B. Semonov, The Implementation of a Commit Messages Filter for Software Version Control Systems, in: The 9th International Conference on Control and Optimization with Industrial Applications, 2024, pp. 175–179.
[8] N.V. Otten, How To Use Text Normalization Techniques In NLP With Python [9 Ways, 2023. URL: https://spotintelligence.com/2023/01/25/text-normalization-techniquesnlp/.
[9] TensorFlow, TensorFlow, 2019. URL: https://www.tensorflow.org/.
[10] Loshchilov I., Hutter F., Decoupled weight decay regularization, in: Proceedings of the ICLR, 2019.
[11] N.K. Nissa, Text Messages Classification using LSTM, Bi-LSTM, and GRU, 2022. URL: https://nzlul.medium.com/theclassification-of-text-messages-using-lstm-bi-lstm-and-gruf79b207f90ad.
[12] Scikit-Learn, scikit-learn: machine learning in Python — scikit-learn 0.16.1 documentation, 2019. URL: https://scikit-learn.org/.
[13] Salesforce, Salesforce/codet5-small, 2021. URL: https://huggingface.co/Salesforce/codet5-small.
[14] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, S. C. H. Hoi, CodeT5+: Open Code Large Language Models for Code Understanding and Generation, 2023. URL: https://arxiv.org/abs/2305.07922.
[15] S. Longpre, L. Hou, T. Vu, A. Webson, H. W. Chung, Y. Tay, D. Zhou, Q. V. Le, B. Zoph, J. Wei, et al., The Flan Collection: Designing Data and Methods for Effective Instruction Tuning, 2023. URL: https://arxiv.org/abs/2301.13688.
[16] J. Han, W. Buntine, E. Shareghi, Reward Engineering for Generating Semi-structured Explanation, 2024. URL: https://arxiv.org/abs/2309.08347.
[17] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, et al., HuggingFace's Transformers: State-of-the-art Natural Language Processing, 2020. URL: https://arxiv.org/abs/1910.03771.
[18] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, M. Auli, fairseq: A Fast, Extensible Toolkit for Sequence Modeling, 2019. URL: https://arxiv.org/abs/1904.01038.
[19] S. Watanabe, T. Hori, S. Karita, T. Hayashi, J. Nishitoba, Y. Unno, N. E. Y. Soplin, J. Heymann, M. Wiesner, N. Chen, et al., ESPnet: End-to-End Speech Processing Toolkit, 2018. URL: https://arxiv.org/abs/1804.00015.
[20] Google, Colaboratory – Google, 2024. URL: https://research.google.com/colaboratory/faq.html.
[21] K. Papineni, S. Roukos, T. Ward, W.-J. Zhu, Bleu: a Method for Automatic Evaluation of Machine Translation, in: P. Isabelle, E. Charniak, D. Lin (Eds.), Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, Philadelphia, Pennsylvania, USA, 2002, pp. 311–318. doi:10.3115/1073083.1073135.
[22] C.-Y. Lin, ROUGE: A Package for Automatic Evaluation of Summaries, in: Text Summarization Branches Out, Association for Computational Linguistics, Barcelona, Spain, 2004, pp. 74–81. URL: https://aclanthology.org/W04-1013/.
[23] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, Y. Artzi, BERTScore: Evaluating Text Generation with BERT, 2020. URL: https://arxiv.org/abs/1904.09675.
[24] T. Sellam, D. Das, A. P. Parikh, BLEURT: Learning Robust Metrics for Text Generation, 2020. URL: https://arxiv.org/abs/2004.04696.