Fine-Tuning Transformer Models for Commit Message Generation and Autocompletion

Martin Miksik

m.miksik@student.utwente.nl

University of Twente

Enschede, The Netherlands

ABSTRACT

Commit messages provide insight into the developer's intentions and motivations — a fundamental source of information in the exceptionally collaborative discipline of software development. To assist the process of commit message writing, we fine-tune two transformer models for commit message generation and integrate them into popular code editors.

We 1) collect and publish a dataset of commit message and patch pairs for 6 different programming languages, 2) fine-tune two generative language models for commit message autocompletion and generation tasks, and 3) provide integration for these models with popular code editors (IntelliJ, VSCode). Lastly, we show that on the test dataset, our fine-tuned models perform 2x and 10x times better than the base models for the completion and generation tasks, respectively.

KEYWORDS

commit messages, source version control, documentation, CodeBERTa, CodeT5, transfer learning, git

1 INTRODUCTION

Software maintenance accounts for the majority of the software system lifecycle and the bulk of the software developer job consists of reading and understanding code [15, 22]. Thus, good documentation is crucial for software maintenance and longevity[2, 16]. This paper focuses on code documentation in the form of commit messages.

In a version control system (VCS) a commit represents an isolated change to the code base [3]. They are composed of code (or other) artefact and textual logs describing *what* changed and *why*—commit message [3, 23]. It should facilitate the review process and help the other contributors to understand the impact of the changes [23]. For example see figure 1.

Tian et al. states that in long-lived projects commit messages might be the only existing or reliable source of documentation [23]. Buse and Weimer stipulate that writing good commit messages is, however, a time-consuming process and in their research one-third of analyzed commit messages included inaccurate information [3].

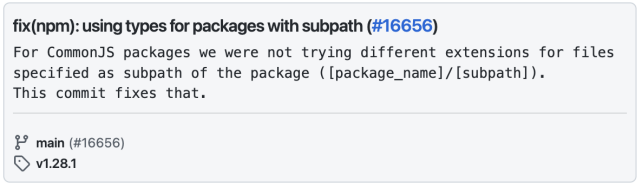


Figure 1: Example of commit message from DenoJS repository. The headline is in bold and provides an answer to the *what* question, while the body gives the answer to *why* [1]

The problem is especially pressing in open source software where developers are geographically dispersed and come from a variety of cultures and educational backgrounds, possibly further diversifying the quality and consistency of commit messages [2, 23]. Ko et al. found that even in colocated teams the design rationale (a.k.a *why*) is the most discussed topic [14]. Furthermore, geographically distributed teams also face the challenge of relying primarily on written communication [2].

Many tools were developed to aid the developers with the process of writing code. Namely, code completion and code generation have experienced leap advancement with the advent of machine learning algorithms. For instance, GitHub Copilot—a publicly available version of the Codex model [5]—is a novel tool that is capable of translating documentation strings into code and vice versa.

Yet, tooling to assist commit message writing is missing. While there exists previous research on this topic done by Cortes-Coy et al., Huang et al., Liu et al., we aim to push the status quo by fulfilling the following goals:

- **Goal 1:** Support multiple programming languages,
- **Goal 2:** Leverage pre-trained transformer-based models, to circumvent the necessity for large training datasets,
- **Goal 3:** Developing convenient integration with modern code editors (e.g. VSCode, IntelliJ), in order to enable simpler adoption and frictionless developer experience.

We aim to achieve these goals by answering the following research questions(RQ):

[1] deno/commit/1416713cb3af8a952b1ae9952091706e2540341c

TScIT 37, July 8, 2022, Enschede, The Netherlands

- **RQ 1:** How does the accuracy of CodeBERTa compare to the accuracy of fine-tuned CodeBERTa on the autocompletion task?
- **RQ 2:** How does fine-tuned CodeT5 compare to base CodeT5 on commit message generation task?

2 RELATED WORK

We identified two diverging approaches to tackle the commit message generation and autocompletion: (1) *History* and (2) *Learning* based.

The *history-based* generation involves comparing the current changes in a code repository to previous commits. The system identifies the previous commits that are most similar to the current changes and then suggests the commit messages of those previous commits as a starting point for the new commit message. Examples of this work are ChangeDoc[9] and ChangeScribe [17].

This approach has several limitations: the suggested commit messages do not include any information specific to the current changes (e.g. file name), and the resulting quality depends on the commit message consistency and commits history size. On the other hand, the benefit of this approach is that it is programming language agnostic, and thus, can be used in any software stack.

The *learning-based* approach in theory overcomes the limitations of the history-based generation, although it faces the problem of being limited to the selection of programming languages the model was trained on. Past work in this area includes *PtrGNCMsg*—a custom *recurrent-neural-network* [18] which pushed the stat-of-the-art in 2019 by allowing for out-of-vocabulary words to be part of the commit message, *CoRec*—long short-term memory architecture that attempts to remove the bias towards high-frequency words [25] and *CommitBERT*—transformer encoder-decoder model, where instead of a custom encoder, CodeBERTa is used [13].

PtrGNCMsg and *CoRec* models support only input in Java. CommitBERTa currently supports Python and Javascript[2]. Furthermore, the models, except for CommitBERT, are not publicly available.

Only ChangeScribe [17] provides integration with some IDE (*Eclipse*). CommitBERT offers *command-line* integration [13].

Lastly, all previously listed approaches focus on commit message generation exclusively and do not provide *autocompletion* for a partially written message.

3 SELECTION OF COMMIT MESSAGE STYLE

Chacon and Straub in their book, which is now part of the official Git documentation, recommend that the commit message starts with a single line of 50 characters or less describing the changes, followed by one blank line and a detailed explanation. Furthermore, the subject line should be written in an imperative form (i.e.: "Fix nullptr exception" and not "Fixes nullptr exception" or "Fixed nullptr exception") [4].

Jiang and McMillan commit analysis of 1 000 most stared Java repositories (about 2 million commits) showed that nearly 47% of them start with an imperative verb[3] [12].

Some repositories, however, use an alternative format such as *Conventional Commit* [1]. In this style, the imperative verb is proceeded by commit *type* (e.g. fix, chore, feat) and the scope of the changes in brackets. Figure 2 shows a real word example of such a commit.

```
✓ fix(docs): Add German language link inside certifications (#48608)
Update index.md

Added German language in live certifications section.
```

Figure 2: Example of the *Conventional Commit* message format[4]

For our research, we decided to follow the official commit message style, because it represents the most basic and, thus, the most universal form of the subject line. Additionally, tools to help users write conventional commit prefixes and scope already exist[5].

4 DATASET PREPARATION

Models we use during finetuning were pre-trained on *CodeSearchNet Challange* Dataset [10], which includes code snippets in 6 programming languages—Python, Java, Javascript, Ruby, Go, PHP. We, therefore, limited our dataset to these 6 programming languages as well.

Repositories are selected based on their high number of stars, which can be thought of as similar to likes on Facebook. For instance, repositories such as facebook/react or nicolargo/glances are scraped. Furthermore, a few repositories are cherry-picked based on the commit quality and/or count. We published the final dataset and several checkpoints on 🤗mamiksik/processed-commit-diffs. The description also includes the full list of scraped repositories.

4.1 Collecting Raw Data

The data are collected using GitHub Rest API (version 2022/11/28), which is subject to a limit of 5 000 requests per hour. In order to facilitate the scraping process efficiently, it is divided into three phases. Phase **A** involves a bulk collection of commit messages, with approximately 100 messages being obtained per request. In phase **B**, commit messages that do not meet the requirements outlined in section 3 are eliminated. Supplementary filters are added to increase the dataset quality; that includes removing *housekeeping*, *bot-generated* commits or non-specific commits messages. Lastly, in phase **C**, commit diffs are gathered.

Phase B is detailed in figure 3. Subject length is calculated after striping *conventional commit prefix* and references to *GitHub issues* (e.g. #10242). Commit messages composed of ≤ 2 words are also removed since they are not specific enough (e.g. Fix main.py).

[2] https://github.com/graykode/commit-autosuggestions/blob/2bc18fdbdcc38d3e5b77fc3471fcd860b3057e89/README.md

[3] The analysis was done using Stanford CoreNLP library, thus, the true proportion might differ.

[4] freeCodeCamp/commit/d57da28c4fd8b3223daf17ae7737fe726ed45c18

[5] For example lppedd/idea-conventional-commit

Detecting a verb-object combination is done using *part-of-speech tagging*, however, it is a non-trivial task, that is impractical to implement using rule-based algorithms. Therefore, we tested two popular natural language processing libraries SpaCy and Stanza. In our testing, both SpaCy and Stanza failed to always accurately identify verb-object combinations in commit messages[6]. On that account, to avoid rejecting valid commit messages, we employed both SpaCy and Stanza, accepting the commit if either tool identified a verb-object combination.

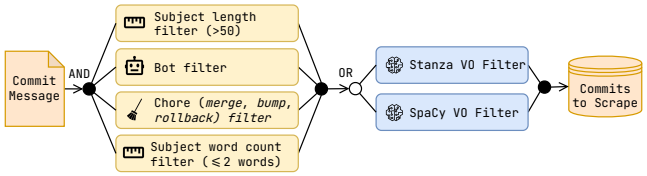


Figure 3: Commit message filtering pipeline based on requirments set in 3. Rule-based filters are yellow. ML filters are blue

4.2 Dataset cleanup

Neither scraped repository is strictly monolingual (e.g. it contains config files), these files could confuse the model during training. Thus, commits containing unsupported file types are excluded.

For each file, the *change* identifiers are replaced by special tokens [*ADD*] and [*DEL*], respectively. Unmodified lines are prefixed with [*IDE*] token. The file path is appended to the top of the patch and prefixed with a special token [*PATH*][7]. If the file was newly added, removed or renamed, the path is prefixed with the corresponding change identifier token. The *chunk* identifier is removed. Lastly, all patches are concatenated into a single string (See the right part of the figure 4).

Processing of the commit message subjects consists of several tasks aimed at removing project-specific information such as issue id, conventional commit prefixes or capitalization (See the left part of the figure 4).

The final dataset is split into train (80 %), validation (10 %) and test (10 %) subsets.

4.3 Dataset overview

We collected commit messages for 693 195 commits, of which 77 840 passed through the filtering pipeline (Table 1). Examples of accepted and rejected messages can be found in the table 2.

135 988 commits are identified as being related to maintaining the project, such as version bump, rollback or merge commits. Another 5 189 are classified as bot-generated.

The subject line of 235 518 commits is longer than 50 characters, and 235 518 commit messages are composed of 2 or fewer words.

[6]It is likely that neither of the training datasets for SpaCy and Stanza included commit messages

[7]Including the file path leaks the programing language information to the model since it includes the file extension.

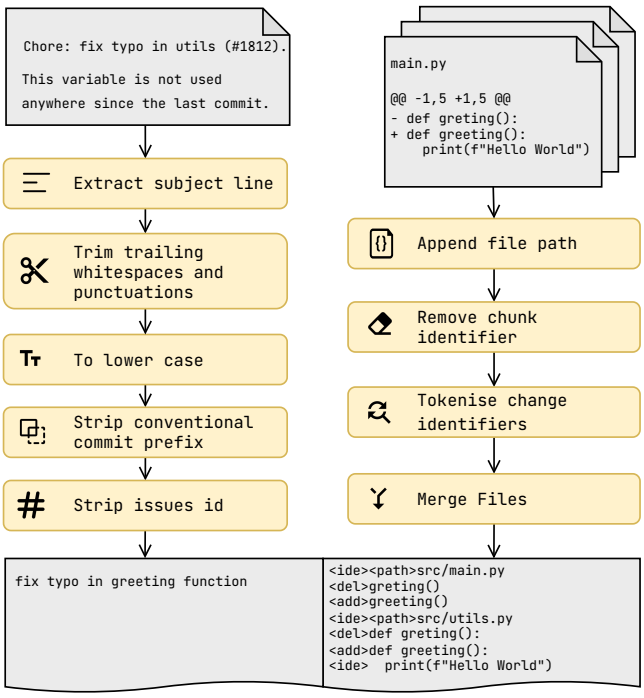


Figure 4: Data preprocessing pipeline

Table 1: Number of fetched commit diffs per language in ascending order

Java	Go	PHP	Ruby	Python	Javascript	Total
3 045	5 943	11 264	13 981	17 065	26 542	**77 840**

The interquartile range in figure 5 shows that half of the commit messages are between 29 and 55 characters long.

The most used verbs are charted in figure 6. VO-Filter rejected 362 202 commits. Spacy VO-Filter and Stanza VO-Filter disagreed in 186 272 cases.

5 ARCHITECTURE SELECTION

Transformer [24], first introduced in 2017, has become the dominant natural language processing architecture in recent years [28]. They provide 3 key innovations for NLP tasks over Recurrent Neural Networks (RNN) previously used for NLP [24].

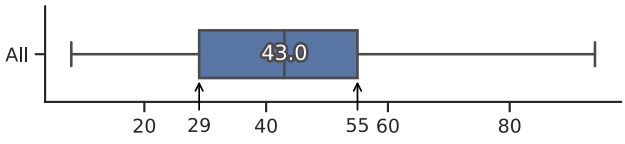


Figure 5: Boxplot of a commit message length in characters. The interquartile range is 26 characters.

Table 2: Examples of accepted and rejected commit messages

Rejected	Message	Note
No	Add test fixture for financial sample	—
	Update ci config for new type checking flow	VO-Filters (Stanza=True, SpaCy=False)
Yes	types: work around backburner runtime paths shenanigans	VO-Filters (Stanza=False, SpaCy=False)
	Correct FEATURES.md	Word count ≤ 2
	Bump version for beta release of v3	House keeping task

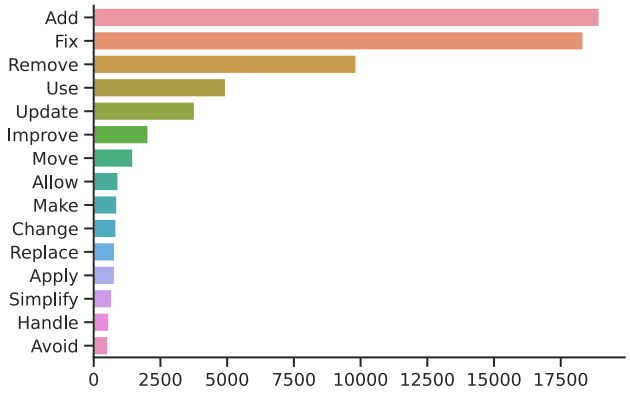


Figure 6: Top 15 verbs used for commit subject (case insensitive)

(1) Contextual Embeddings: Transformers can internally generate word (token) embeddings based on the context of the entire text (thousand and more words). While a model with bi-directional RNN technically has the same ability, it suffers from vanishing gradients. Thus in practice is unable to condition words' embedding on the next/previous sentence, let alone a different paragraph. Older techniques using classical ML pre-trained directly on word embeddings like Word2Vec, Glove or FastText use no context [24].

(2) Attention Mechanism: The attention (and self-attention) provides the network with the relationship information among words. That enables the network to focus on the most relevant information in the input text (which is context dependent)[24].

(3) Reduction in sequentiality: The model architecture eliminates recurrent steps required in RNN, which greatly increases the possibility to parallelize (and thus accelerate) both learning and inference.[8] [24].

The originally proposed architecture consists of Encoder and Decoder. Encoders contain self-attention layers, they are bi-directional (for each token they can attend both past and consecutive tokens) [24]. Their output consists of a vector sequence of the same length as input [26]. These vectors carry the contextual information about each input word (i.e. each vector encodes both the information about the input token and surrounding tokens) [19].

[8]Recurrent layer requires $O(n)$ sequential steps, whereas self-attention layer requires $O(1)$ sequential steps [24]

Decoders are auto-regressive; which means that generated tokens are fed back to the model to generate the next token. For that reason, decoders can only attend preceding tokens (See figure 7) [24]. The decoder also incorporates a cross-attention layer, thus, the decoder can condition its output based on the output of the encoder [26].

The combination of encoder and decoder allows the model to develop a separate representation of input and output tokens [26]. Hence, Encoder-Decoder architecture is suitable for tasks like machine translation or summarization. An example of encoder-decoder architecture is T5 (Text-To-Text Transfer Transformer) [21]. Thus, we chose the encoder-decoder model, for the goal of commit message generation, called *CodeT5* [27].

On the other hand, models like BERT are encoder-only. Since encoders are bi-directional, this architecture is suitable for tasks requiring a good contextual understanding of the input; such as grammar checking, sentiment analysis or corrupt token replacement [7]. Hence, for the task autocompletion, we chose the encoder-only model *CodeBERTa* [8].

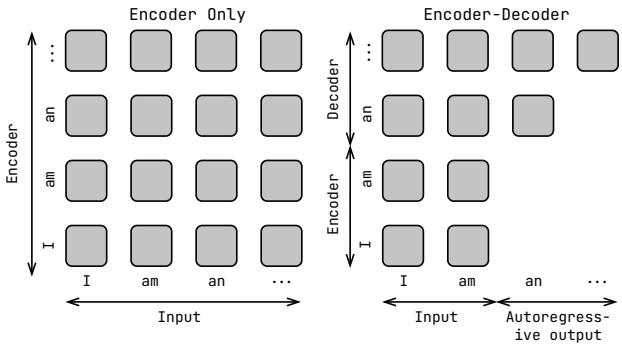


Figure 7: Depiction of which tokens can be attended in encoder-only and encoder-decoder architecture

6 THE AUTOCOMPLETION MODEL

CodeBERTA is a pre-trained model published by Microsoft [8]. It follows the RoBERTa architecture [19], which is an optimization of encoder-only BERT transformer [7]. CodeBERTa model is identical to the $\text{RoBERTa}_{\text{base}}$. It has 12 hidden layers and 12 attention heads. The size of hidden layers is 768 and the attention head size is 64 [19]. CodeBERTa consists of 125M parameters [8].

The two pre-train objectives were Masked Language Modeling (MLM) and Replaced Token Detection (RTD) [8]. In MLM, percentage[9] of tokens in input is substituted by special $[MASK]$ token, and the model must predict appropriate token for $[MASK]$. In RTD, random tokens are swapped for different tokens and the model must detect which tokens were swapped [8].

CodeBERTA was pre-trained on the dataset from *CodeSearchNet Challenge* [10]. Both standalone code snippets and natural language (NL) - programming language (PL) pairs were used for pretraining, and the programing language of the input was not explicitly leaked to the model [8].

CodeBERTa uses WordPiece text encoding rather than Byte pair encoding used in RoBERTa [8, 19].

6.1 Input/Output representation

Our input consists of a commit message (NL) and a patch (PL) pair. It has the following shape $[CLS], d_1, d_2, d_n, [SEP], [MSG], m_1, m_2, m_k, p_1, p_2, p_l, [EOS]$ where where $d_{1..n}$ is tokenized patch, $m_{1..k}$ is tokenized commit message and $p_{1..l}$ is padding. Lastly, $n + k + l = 512$. In a case $n + k > 512$ then $l = 0$ and the patch is truncated so that $n + k = 512$. At least one in $m_{1..k}$ tokens must be $[MASK]$.

The output of the model is a dictionary with predicted tokens as keys and model confidence as associated values.

6.2 Finetuning

We follow the MLM and RTD objectives that CodeBERTa has originally trained on [8], however, rather than masking (or replacing) tokens in both parts of the input sequence, only the tokens corresponding to the commit message are masked/replaced. On that account, during the training, the model should focus exclusively on learning the autocompletion of commit messages. During the pretraining of CodeBERTa, 15% of tokens are masked [8]. We increased the masking ratio to 50 % since commit messages are comparably short. For training, the masked sequence is passed as input to the model, whereas the original sequence is passed as a label (See figure 8).

Input	<ide><path> src/main.py <ide>def main(): <del> name = "John" <ide> print("Hello World!") <msg>Remove <mask> variable <mask> main
Label	<ide><path> src/main.py <ide>def main(): <del> name = "John" <ide> print("Hello World!") <msg>Remove unused variable in main

Figure 8: Example of CodeBERTa training input and label

Live demo is located at 🤗mamiksik/commit-message-autocomplete

[9]In RoBERTa tokens are dynamically masked every epoch, while in BERT model masking happens once before the training starts [7, 19].

7 THE COMMIT MESSAGE GENERATION MODEL

CodeT5 is an encoder-decoder model based on the T5 architecture from Salesforce Research [21, 27]. CodetT5 introduces a few new PL-specific pre-training objectives that aim to improve its performance on code generation and summarization. Several checkpoints of the CodeT5 model were released. We use the $\text{CodeT5}_{\text{base multi sum}}$, which is a version of $\text{CodeT5}_{\text{base}}$ fine-tuned on code summarization task on the *CodeSearchNet Challenge* [10].

The model has 220 million parameters and is composed of 12 layers. Each layer has a hidden state size of 768. The model also uses multi-head attention, with 12 heads.

7.1 Input/Output representation

For CodeT5, the input is the commit patch (PL) and the label is the commit message (NL). The patch and message are limited to 1 024 and 128 tokens, respectively. Both are padded or truncated to their maximal length. The model should be neither rewarded nor punished for predicting padding tokens. Thus, they are replaced by -100, which excludes it from the loss calculation.

The output is the top 5 predictions found using *beam-search* with 7 beams and conditioned by a minimal length of 4 tokens and a maximal length of 128 tokens.

7.2 Finetuning

The objective for finetuning the CodeT5 model is summarization[10]. The training process utilizes Pytorch Lighting and the AdamW optimizer. A learning rate of $5e-5$ is employed, with 1 000 warm-up steps. The training batch size and accumulate gradient steps are both set to 16. The training loop runs on 2 Nvidia A40 GPUs, which results in an effective batch size of 512. An early stopping mechanism is implemented, where if the evaluation learning rate ceases to decrease for three consecutive epochs, the process will terminate. No fixed minimum or a maximum number of epochs is specified.

Furthermore, we monitor the learning progress using the smoothed BLEU-4 [20] score. BLEU-4 is commonly used to evaluate machine translations [5, 8, 27]. BLEU-4 algorithm counts the number of matching n-grams (from 1 to 4). The score is bounded: $0 \leq score \leq 1$ where 0 means no matching n-grams and 1 represents a perfect match.

The definition of BLEU-4 is as follows

$$BLEU\text{-}4 = BP * \exp(\sum_{n=1}^{4} \frac{1}{4} \log p_n)$$

, where BP stands for brevity penalty; punishes translations that are too short compared to the label, p_n stands for *modified unigram precision*; fixes the issues of word repetition for unigram precisions[11]

[10]This goal could be also interpreted as a translation problem. From the *language of code* to the *language of commit messages*

[11]For example, source sequence *"the cat"* translated as *"the the"* would achieve 2/2 precision, while it only achieves 1/2 *modified unigram precision.* [20]

Live demo is located at 🤗mamiksik/commit-message-generator. A few examples of generated messages can also be found in the appendix.

8 INTEGRATED DEVELOPMENT ENVIRONMENT INTEGRATION

We develop an integration for the JetBrains Platform (e.g. PyCharm, IntelliJ, PHPStorm) and for Visual Studio Code (VSCode). The plugin seamlessly integrates with the built-in commit interface. For instance, predictions are only generated based on changes selected in the user interface rather than the direct output of the '$ git diff' command.

The predictions are presented to the user in the integrated autocomplete dialogue, which can be activated by pressing 'Ctrl+Space' on Linux and Windows or 'Cmd+Space' on a Mac. New predictions are also generated every time the typed message ends with an empty space. The plugin for JetBrains Platform is available in the Jebrains Marketplace under the name *Parrot - AI Commit Message Autocomplete* or at mamiksik/parrot-intellij, and the VSCode plugin is available at mamiksik/parrot-vscode

The plugin is complemented by a Hypertext Transfer Protocol (HTTP) server that can be run locally (in Python venv or Docker Container), which takes care of providing the predictions. The client-server architecture mitigates resource waste, as a single server can service multiple running code editor instances. It is located at mamiksik/parrot-server.

9 EVALUATION

In this section, the empirical results of our model's effectiveness are presented. Each model is tested using quantitative means. The *Commit Message Generation* model is also evaluated qualitatively.

9.1 Auto-completion Model

The accuracy metric is used to quantitatively evaluate the autocompletion (i.e. the masked token replacement task). Accuracy provides a clear insight into the percentage of masked tokens that the model can accurately recover from the given context. The fine-tuned model is compared to the $\text{CodeBERTa}_{\text{base}}$ and to $\text{RoBERTa}_{\text{base}}$ models.

Table 3: Accuracy (in %) on the test dataset for the autocompletion model ($\overline{X}$ is weighted average)

Model	$\overline{X}$	Java	Go	PHP	JS	Ruby	Python
RoBERTa	37.4	32.1	39.7	30.6	36.0	38.7	41.3
CodeBERT	34.8	34.3	33.5	30.6	33.2	35.7	37.0
Parrot	**64.7**	**67.4**	**63.3**	**64.9**	**64.3**	**64.6**	**64.6**

The results in table 3 show that the overall performance of our model after fine-tuning for the mask token replacement task is almost double the baseline. Furthermore, the performance is equivalent across all supported programming languages.

The model evaluated in table 3 can be found at 🤗CodeBERTa-commit-message-autocomplete revision *#cf7f2e5*.

9.2 Commit Message Generation Model

In quantitative testing, the fine-tuned model was compared to the $\text{CodeT5}_{\text{base multi sum}}$ model[12]. The BLEU-4 metric is computed on the whole test dataset as well as, for each programing language separately. The combined score shows that the fine-tuned model outperforms the base model by over 10 % (Table 4). The fine-tuned model performs best for input in Go and worst in input in PHP. This is a surprising result considering that the number of examples for PHP was almost double (see table 1). It would suggest that the scraped commit messages for PHP are of worse quality, since the original model performs better on PHP summarization than on Go summarization [27].

The model evaluated in table 4 can be found at 🤗T5-commit-message-generation revision *#fb08d01*.

For qualitative testing, 10 examples are randomly sampled from the *test* data subset and commit messages are generated for them. We asked 2 experienced developers (+3 years of experience) to evaluate the generated messages based on the patch and compare them to the original commit message (ground truth) as well. Next, the developers are asked to write a short summary of their impressions.

Tester 1:

> The tool seems to be very good at identifying what parts of code the commit touches, even if the fix has something to do with responsivity, performance, or other abstract concepts. It does sometimes blunder the exact nature of the change, however, labelling general fixes as "fixing typos" or getting confused about whether something was added or removed. The commit messages generated follow conventions and seem appropriate for usage in real-world scenarios.

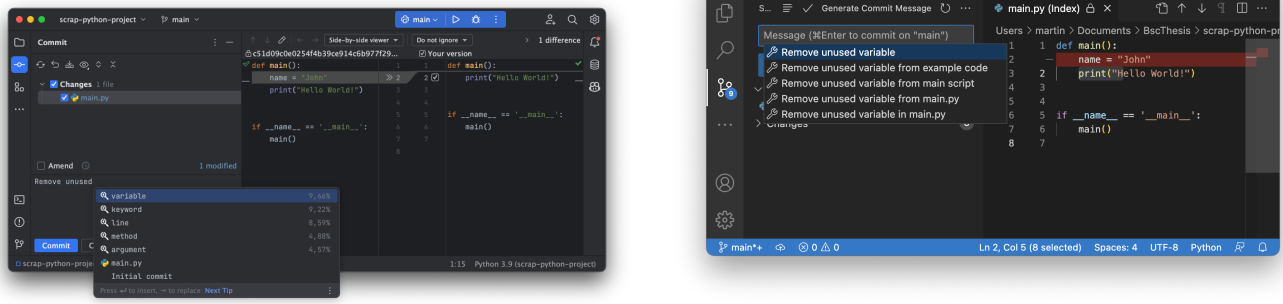
Tester 2:

> The generated commit messages are well-structured and follow the standard message conventions. While not all messages captured the changes accurately, they could work as a good starting point. On the other hand, the generated messages for simple code changes could be used without any modification.

Table 4: Results of BLEU-4 metric (in %) on the test dataset. We implemented the T5 training loop in Huggigface trainer (hf) and Pytorch Lighting (pl) ($\overline{X}$ is weighted average)

Model	$\overline{X}$	Java	Go	PHP	JS	Ruby	Python
CodeT5	0.32	0.29	0.41	0.34	0.27	0.29	0.51
$\text{Parrot}_{\text{hf}}$	8.88	2.31	8.26	3.57	8.79	8.62	10.13
$\text{Parrot}_{\text{pl}}$	**11.54**	**4.75**	**14.68**	**3.99**	**12.64**	**10.08**	**13.05**

[12]We contacted the authors of PtrGNCMsg with an inquiry about the model and received no answer. Authors of CoRec responded, however, they have deleted their model, thus they were unable to provide it for testing.



(a) IntelliJ, Commit message autocompletion by the message generation model

(b) VSCode, Commit message prediction by the message generation model

Figure 9: Integration with VSCode and IntelliJ

Table 5: Inference latency. Sampled 1 000 times on M1 Pro (CPU) and Nvidia GPUs (Titan X, A40).

	Model Size	M1 Pro	Titan X	A40
Autocompletion	± 500 MB	333ms	32ms	15ms
Generation	± 890 MB	1.56s	311ms	331ms

10 FUTURE WORK

We found the latency of commit message generation to be the biggest challenge during our research (see table 5). On that account, future research could focus on optimizing the model for faster inference. One method to do that is *quantization*[13]; which is a process of replacing the floating point weights with integer arithmetics. This results in both smaller model sizes and faster inference [11]. Another optimization could involve exporting the model for ONNX Runtime[14], which is able to better utilize platform-specific ML accelerators (Such as CoreML on macOS or DirectML on Windows).

Furthermore, we suggest developing a model for commit accuracy classification, which could be integrated into IDE in a form of a traffic light. That would signal to the developer that their commit message accurately captures staged changes.

11 CONCLUSION

In this paper, we present two fine-tuned transformer models for commit message completion and generation. The autocompletion model is based on the encoder-only model CodeBERTa, while the message generation model is based on the encoder-decoder CodeT5 model. For fine-tuning, we collect (and publish) a dataset of about 77K samples of commit message and patch pairs in 6 different programming languages. The finetuned models achieve 2x and 10x better performance compared to the base models respectively. Lastly, we develop an extension for both the IntelliJ platform and Visual Studio Code, that integrates our models into the native git user interface in the given code editors.

On that account, we fulfilled the three goals we set in the beginning. That is: (1) Support multiple programming languages, (2) Leverage pre-trained transformer-based models, to circumvent the necessity for large training datasets, (3) Developing convenient integration with modern code editors (e.g. VSCode, IntelliJ), in order to enable simpler adoption and frictionless developer experience.

[13]Using tools such as Huggingface Optimum or Intel Neural Compressor
[14]https://onnxruntime.ai

REFERENCES

[1] [n. d.]. Conventional Commits. https://www.conventionalcommits.org/en/v1.0.0/

[2] Rana Alkadhi, Manuel Nonnenmacher, Emitza Guzman, and Bernd Bruegge. 2018. How do developers discuss rationale? *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings* 2018-March (4 2018), 357–367. https://doi.org/10.1109/SANER.2018.8330223

[3] Raymond P L Buse and Westley Weimer. 2010. Automatically Documenting Program Changes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. 33–42. https://doi.org/10.1145/1858996.1859005

[4] Scott Chacon and Ben Straub. [n. d.]. Git - Contributing to a Project. https://www.git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project#_commit_guidelines

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. (2021). https://www.github.com/openai/human-eval.

[6] Luis Fernando Cortes-Coy, Mario Linares-Vasquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On automatically generating commit messages via summarization of source code changes. In *Proceedings - 2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014*. Institute of Electrical and Electronics Engineers Inc., 275–284. https://doi.org/10.1109/SCAM.2014.14

[7] Jacob Devlin, Ming Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference* 1 (10 2018), 4171–4186. https://doi.org/10.48550/arxiv.1810.04805

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *Findings of the Association for Computational Linguistics Findings of ACL: EMNLP 2020* (2020), 1536–1547. https://doi.org/10.18653/V1/2020.FINDINGS-EMNLP.139

[9] Yuan Huang, Nan Jia, Hao-Jie Zhou, Xiang-Ping Chen, Ieee Zi-Bin Zheng, Senior Member, and Ming-Dong Tang. 2020. Learning human-written commit messages to document code changes. *JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY* 35, 6 (2020), 1258–1277. https://doi.org/10.1007/s11390-020-0496-0

[10] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Github Miltiadis, and Allamanis Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv preprint arXiv:1909.09436* (9 2019). https://doi.org/10.48550/arxiv.1909.09436

[11] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (12 2017), 2704–2713. https://doi.org/10.48550/arxiv.1712.05877

[12] Siyuan Jiang and Collin McMillan. 2017. Towards Automatic Generation of Short Summaries of Commits. In *IEEE International Conference on Program Comprehension*. IEEE Computer Society, Buenos Aires, Argentina, 320–323. https://doi.org/10.48550/arxiv.1703.09603

[13] Tae Hwan Jung. 2021. CommitBERT: Commit Message Generation Using Pre-Trained Programming Language Model. *NLP4Prog 2021 - 1st Workshop on Natural Language Processing for Programming, Proceedings of the Workshop* (5 2021), 26–33. https://doi.org/10.48550/arxiv.2105.14242

[14] Andrew J. Ko, Robert DeLine, and Gina Venolia. 2007. Information needs in collocated software development teams. In *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, USA, 344–353. https://doi.org/10.1109/ICSE.2007.45

[15] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering* 32, 12 (12 2006), 971–987. https://doi.org/10.1109/TSE.2006.116

[16] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. 2003. How Software Engineers Use Documentation: The State of the Practice. *IEEE Software* 20, 6 (11 2003), 35–39. https://doi.org/10.1109/MS.2003.1241364

[17] Mario Linares-Vasquez, Luis Fernando Cortes-Coy, Jairo Aponte, and Denys Poshyvanyk. 2015. ChangeScribe: A Tool for Automatically Generating Commit Messages. *Proceedings - International Conference on Software Engineering* 2 (8 2015), 709–712. https://doi.org/10.1109/ICSE.2015.229

[18] Qin Liu, Zihe Liu, Hongming Zhu, Hongfei Fan, Bowen Du, and Yu Qian. 2019. Generating commit messages from diffs using pointer-generator network. *IEEE International Working Conference on Mining Software Repositories* 2019-May (5 2019), 299–309. https://doi.org/10.1109/MSR.2019.00056

[19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, Veselin Stoyanov, and Paul G Allen. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv preprint arXiv:1907.11692* (7 2019). https://doi.org/10.48550/arxiv.1907.11692

[20] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02* (2002), 311–318. https://doi.org/10.3115/1073083.1073135

[21] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21 (10 2019), 1–67. https://doi.org/10.48550/arxiv.1910.10683

[22] Zephyrin Soh, Foutse Khomh, Yann Gael Gueheneuc, and Giuliano Antoniol. 2013. Towards understanding how developers spend their effort during maintenance activities. In *Proceedings - Working Conference on Reverse Engineering, WCRE*. Koblenz, Germany, 152–161. https://doi.org/10.1109/WCRE.2013.6671290

[23] Yingchen Tian, Yuxia Zhang, Klaas Jan Stol, Lin Jiang, and Hui Liu. 2022. What Makes a Good Commit Message? *Proceedings - International Conference on Software Engineering* 2022-May (2022), 2389–2401. https://doi.org/10.1145/3510003.3510205

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *Advances in Neural Information Processing Systems* 2017-December (6 2017), 5999–6009. https://doi.org/10.48550/arxiv.1706.03762

[25] Haoye Wang, David Lo, John Grundy, Xin Xia, Qiang He, and Xinyu Wang. 2021. Context-Aware Retrieval-based Deep Commit Message Generation. *ACM Reference Format* (2021). https://doi.org/10.1145/nnnnnnn.nnnnnnn

[26] Thomas Wang, Adam Roberts, Daniel Hesslow, Teven Le Scao, Hyung Won Chung, Iz Beltagy, Julien Launay, Colin Raffel, and Hugging Face. 2022. What Language Model Architecture and Pretraining Objective Work Best for Zero-Shot Generalization? The BigScience Architecture & Scaling Group. https://github.com/bigscience-workshop/architecture-objective.

[27] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*. Association for Computational Linguistics (ACL), 8696–8708. https://doi.org/10.48550/arxiv.2109.00859

[28] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clément Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics (ACL), Online, 38–45. https://doi.org/10.18653/V1/2020.EMNLP-DEMOS.6

A EXAMPLES OF GENERATED COMMIT MESSAGES

```
func (container *Container) Attach(stdin io.ReadCloser,
    ↪ stdinCloser io.Closer, s
   } else {
    _, err = io.Copy(cStdin, stdin)
   }
+  if err == io.ErrClosedPipe {
+   err = nil
+  }
   if err != nil {
    utils.Errorf("attach: stdin: %s", err)
   }
-   // Discard error, expecting pipe error
-   errors <- nil
+   errors <- err
   }()
  }
 }
```

Ground Truth	Ignore errclosedpipe for stdin in container.attach
ChatGPT	Handling io.ErrClosedPipe when copying stdin and sending error value to errors channel
Parrot	Ignore errclosedpipe when attaching stdin

Figure 10: Example 1 (Adapted from test dataset)

```
- a/.github/workflows/pylint.yml
+ b/.github/workflows/codestyle_checks.yml
    - name: Analysing the code with pylint
      run: |
        pylint --rcfile=.pylintrc webapp core
+   - name: Analysing the code with flake8
+     run: |
+       flake8
```

Ground Truth	Add flake8 as GitHub action
ChatGPT	Rename workflow file and add flake8 checks for code style consistency
Parrot	Add flake8 to codestyle_checks.yml

Figure 11: Example 2 for *unsuported* language (YAML) (adapted from a personal private repository)

```
- return JsonLDUtils.jsonLdGetStringList(
- this.getJsonObject(), Keywords.CONTEXT).stream().map(
- JsonLDUtils::stringToUri).collect(Collectors.toList());
+ List<String> contextStrings =
+ JsonLDUtils.jsonLdGetStringList(this.getJsonObject(),
+ Keywords.CONTEXT);
+ return contextStrings == null ? null : contextStrings
+ .stream().map(JsonLDUtils::stringToUri)
+ .collect(Collectors.toList());
```

Ground Truth	fix: Fix potential null pointer exception.
ChatGPT	Fix null pointer exception in context string list processing
Parrot	Handle null in JsonLDUtils.jsonLdGetStringList

Figure 12: Example 3 Source: decentralized-identity/jsonld-common-java

```
+ import io
import logging
import os
import re
def open_maybe_zipped(fileloc, mode='r'):
    """
    _, archive, filename =
        ↪ ZIP_REGEX.search(fileloc).groups()
    if archive and zipfile.is_zipfile(archive):
-       return zipfile.ZipFile(archive, mode=mode)
-       .open(filename)
+       return io.TextIOWrapper(zipfile.ZipFile
+       (archive, mode=mode).open(filename))
    else:
        return open(fileloc, mode=mode)
```

Ground Truth	fix reading from zip package to default to text
ChatGPT	Replace zipfile.open with io.TextIOWrapper for handling zipped files
Parrot	Fix error when reading from zip package

Figure 13: Example 4 (adapted from the test dataset)

```
public function sharedGet($path) {
  $contents = '';

-  $handle = fopen($path, 'r');
+  $handle = fopen($path, 'rb');

  if ($handle) {
    try {
      if (flock($handle, LOCK_SH)) {
-       while (! feof($handle)) {
-         $contents .= fread($handle, 1048576);
-       }
+       clearstatcache(true, $path);
+
+       $contents = fread($handle, $this->size($path) ?: 1);
+
+       flock($handle, LOCK_UN);
      }
    } finally {
      fclose($handle);
```

Ground Truth	fix unix socket
ChatGPT	Changed file open mode for $path from 'r' to 'rb' for binary safe reading and added clearstatcache() and flock() for file locking and unlocking
Parrot	fix unix socket issue

Figure 14: Example 5 (adapted from the test dataset)