From Commit Message Generation to History-Aware Commit Message Completion

Aleksandra Eliseeva
JetBrains Research
Republic of Serbia
alexandra.eliseeva@jetbrains.com

Yaroslav Sokolov
JetBrains
Germany
yaroslav.sokolov@jetbrains.com

Egor Bogomolov
JetBrains Research
Republic of Cyprus
egor.bogomolov@jetbrains.com

Yaroslav Golubev
JetBrains Research
Republic of Serbia
yaroslav.golubev@jetbrains.com

Danny Dig
JetBrains Research
University of Colorado Boulder
United States
danny.dig@jetbrains.com

Timofey Bryksin
JetBrains Research
Republic of Cyprus
timofey.bryksin@jetbrains.com

***Abstract*—Commit messages are crucial to software development, allowing developers to track changes and collaborate effectively. Despite their utility, most commit messages lack important information since writing high-quality commit messages is tedious and time-consuming. The active research on commit message generation (CMG) has not yet led to wide adoption in practice. We argue that if we could shift the focus from commit message generation to *commit message completion* and use previous *commit history* as additional context, we could significantly improve the quality and the personal nature of the resulting commit messages.**

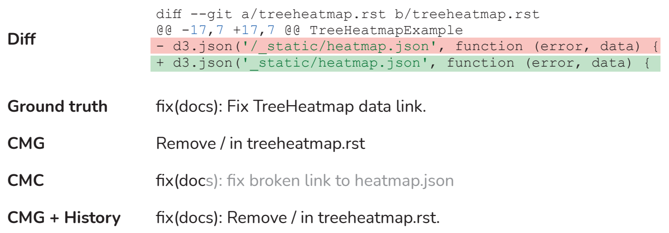
In this paper, we propose and evaluate both of these novel ideas. Since the existing datasets lack historical data, we collect and share a novel dataset called *CommitChronicle*, containing 10.7M commits across 20 programming languages. We use this dataset to evaluate the completion setting and the usefulness of the historical context for state-of-the-art CMG models and GPT-3.5-turbo. Our results show that in some contexts, commit message completion shows better results than generation, and that while in general GPT-3.5-turbo performs worse, it shows potential for long and detailed messages. As for the history, the results show that historical information improves the performance of CMG models in the generation task, and the performance of GPT-3.5-turbo in both generation and completion.

I. Introduction

Whenever a developer commits their work to a version control system, they can write a short comment in a natural language, called a *commit message*. High-quality commit messages can greatly aid software maintenance, as they provide a human-readable overview of what was changed or why and may ease code review or other activities that require the comprehension of changes [1]. In contrast, poor commit messages can negatively affect software defect proneness [2].

Writing a good commit message is tedious and requires extra time and effort from developers. Research shows that a significant amount of commit messages from open source projects lack important information [3] or are even empty [4].

To assist developers, the research community has been working actively on *commit message generation (CMG)* [5]–[19], which is defined as: given the changes made in a commit, generate an appropriate message. Despite numerous technical



Diff	`diff --git a/treeheatmap.rst b/treeheatmap.rst`
	`@@ -17,7 +17,7 @@ TreeHeatmapExample`
	`- d3.json('/_static/heatmap.json', function (error, data) {`
	`+ d3.json('_static/heatmap.json', function (error, data) {`

Ground truth	fix(docs): Fix TreeHeatmap data link.
CMG	Remove / in treeheatmap.rst
CMC	fix(docs): fix broken link to heatmap.json
CMG + History	fix(docs): Remove / in treeheatmap.rst.

Fig. 1. Motivating example for our two ideas for personalizing commit message generation. CMG = standard commit message generation; CMC = commit message completion; CMG + history = commit message generation with commit message history as additional context.

advances in the field, we still do not have wide adoption of CMG in practice, as several practical aspects are not yet solved. Researchers [8], [14], [20] found out that over 50% of messages from existing CMG approaches were inadequate, *i.e.*, semantically irrelevant to the reference messages.

In this paper, we propose two novel approaches to improve the quality of generated commit messages from the standpoint of their personalization: *commit message completion* instead of generation, and *taking the user's previous commit messages into account*. To the best of our knowledge, neither of these approaches has been tested in this context before.

First, we rephrase the problem of commit message generation into *commit message completion*. This way, the already-typed prefix of the commit message helps the model suggest the following relevant tokens. Completion systems for both natural languages [21] and programming languages [22] are now widespread and overall well accepted by end users. In addition, practitioners explicitly name completion as a way they might prefer to use CMG approaches [23]. We hypothesize that a prefix might guide the existing approaches towards more applicable predictions and improved adherence to project conventions, as illustrated in Figure 1.

Second, to aid personalization, we take into account syntactic and stylistic conventions of a particular project or user by *considering the history of their commits* as a part of the model's input. Figure 1 also illustrates this. Such an approach requires two things: **(a)** we must have this *history* saved in the training dataset, and **(b)** such a dataset must be not only large but also *diverse* for the models to learn various possible conventions and realistic commit messages.

We start our work by studying existing CMG datasets. Unfortunately, all of them have the same crucial shortcomings: no saved history and restrictive data filtering. To mitigate this issue, we build a large-scale multilingual dataset that incorporates the best practices from previous works while avoiding the use of filters restricting the representativeness of the data. The dataset is called *CommitChronicle* and contains $\sim$10.7M commits in 20 programming languages from $\sim$12K repositories with permissive licenses. To the best of our knowledge, our dataset is the only one that both provides author metadata and keeps commit history close to the origins.

Since we suggest two separate ideas—reformulating commit message generation to completion and using commit message history—we experiment with all four possible configurations, as shown in Figure 2.

To evaluate the completion setting, we study how different models perform in it. From CMG models, we experiment with CodeT5 [24], RACE [19], and CodeReviewer [25]. Our research demonstrates that metrics scores generally increase when the bigger part of the original message is passed to the context. This finding suggests that completion could be an effective way to apply existing approaches. We also experiment with a large language model GPT-3.5-turbo (also known as ChatGPT) with a simple and straightforward prompt, which demonstrates generally worse results than dedicated CMG models, however, shows potential for longer and more detailed messages. Our findings suggest that the completion setting is significantly easier in practice: for the best model, the average *B-Norm* value grows from 16.9 in the generation setting to 27.2 when the user already typed half of the message.

We evaluate our second idea of considering the commit history and show that the history improves the performance of CMG models for generation (*B-Norm* improves from 15.3 to 16.9), however, the results are conflicting for completion. As for GPT-3.5-turbo, the results improve in both settings. Finally, we study the impact of restrictive data filters by comparing the models on commits that pass all the restrictions and those that do not. Our comparison shows that the results are much better for commits passing all the restrictions than for the commits that pass neither of the filters (*B-Norm* five times higher), indicating that the existing datasets might inflate the results and fail to account for a lot of in-the-wild commits.

Overall, our results show that both ideas—commit message completion and taking into account the history of commits—show potential in specific scenarios and require further research. This paper makes the following contributions:

- A **novel reframing** of commit message generation into completion, its formulation, and an evaluation setup.
- An **approach** of appending the history of commit messages into the model's input for the tasks of commit message generation and completion.
- A comprehensive **analysis** of existing CMG datasets from the standpoint of them keeping commit message history and the restrictiveness of filters, which shows that they have shortcomings from both points of view.
- A **diverse dataset** called *CommitChronicle* with $\sim$10.7 million commits in 20 programming languages, which preserves information necessary for utilizing commit history. We make the dataset publicly available [26].
- A **comparison** of three state-of-the-art CMG models (CodeT5, RACE, and CodeReviewer) and GPT-3.5-turbo from multiple points of view, including generation against completion and history against no history. Our main results indicate that in some cases, completion is simpler than generation, and using history improves the performance of models. All the code for our models and experiments is available online [27].

II. Background

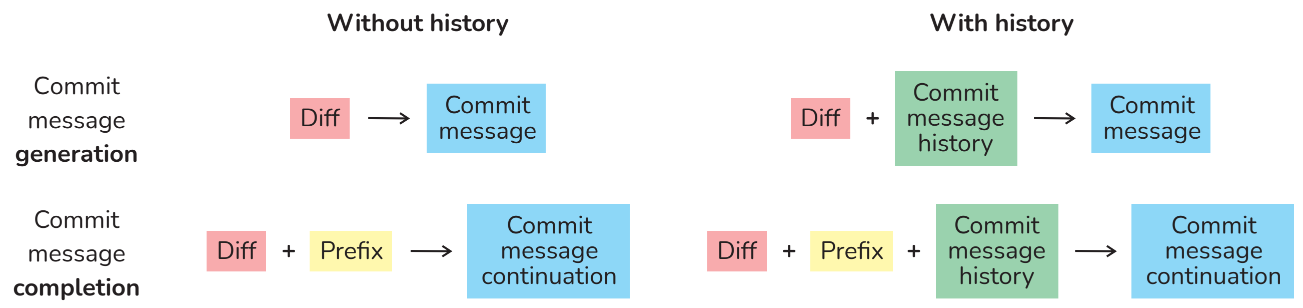
In this section, we give a gentle introduction to the main relevant concepts: commit message generation, completion as used in various domains, and the current uses of large language models (LLMs) in software engineering.

Commit message generation. Modern CMG approaches can be broadly categorized into three groups: language modeling-based, retrieval-based, and hybrid.

CMG may be considered as a task of sequence-to-sequence *language modeling*, analogous to neural machine translation [8] or summarization [9]. In this setting, both input and output are represented as sequences of tokens. During the training phase, given a source sequence $\mathbf{x} = (x_1, \ldots, x_n)$ and a target sequence $\mathbf{y} = (y_1, \ldots, y_m)$, the language model learns a probability distribution $p(y_1, \ldots, y_m | x_1, \ldots, x_n)$ by optimizing the loss function.

Most language modeling-based CMG approaches follow the encoder-decoder structure [8]–[11], [13], [16]. The encoder receives a source sequence $\mathbf{x}$ and produces a vector representation $\mathbf{h} \in \mathbb{R}^d$, where d is a hyperparameter. The decoder receives $\mathbf{h}$ and produces an output sequence $\mathbf{y}$ in an autoregressive manner, *i.e.*, utilizing only previous tokens for each position. Specific neural network architectures considered in prior works include Recurrent Neural Network (RNN) and its modifications [8]–[12], Transformer [13], [16], [19], and Graph Neural Network (GNN) [18].

Unlike language modelling, *retrieval-based* approaches find a commit with the most similar diff in the training set and return a corresponding message. The first such approach is *NNGen* [20]: it uses bag-of-words to represent diffs, retrieves k most similar commits based on cosine similarity, and returns the commit message based on the BLEU score [28] between diffs. Further enhancements to the framework include limiting retrieval to commits from the same repository [29], replacing the metric [15], and considering more complex approaches for representing diffs [30], [31]. Retrieval approaches rely on the presence of similar messages in the training set and may fail due to the unique code identifiers in commit messages.



Without history
With history
Commit message generation
Commit message completion
Diff
Prefix
Commit message
Commit message history
Commit message continuation

Fig. 2. Overview of the four configurations considered in our study and their contents.

Finally, *hybrid* approaches aim to combine the advantages of both language modeling and retrieval. One option is to choose between retrieved and generated messages [32] and the other is to employ the retrieved commits inside the language model to improve the generation quality [19], [23]. At the moment of writing, the recently proposed hybrid approach *RACE* [19] is state of the art in CMG.

Despite all these different approaches, the question of their practical usefulness remains open. In several studies, over 50% of generated messages were rated as low-quality by human experts [8], [14], [20].

Completion. The completion task might be viewed as a simpler version of generation. Rather than generating the full message, completion systems aim to assist a user by suggesting relevant continuation when they start typing. This framework is widely used in practice for diverse domains: writing emails [21] or code completion in IDEs [22], [33]. Popular approaches employ language modelling [21], [22], [34] or ranking, however, the latter is suitable for domains where a limited amount of candidates can be obtained, *e.g.*, code completion [35].

LLMs in software engineering. LLMs are language models scaled up to billions of parameters and billions of tokens in a training set. The majority of recent LLMs are based on the Transformer decoder [36] architecture. A particularly interesting property of LLMs is the in-context learning, *i.e.*, the ability to solve downstream tasks in a few-shot setting (with just several examples passed as the context) or even zero-shot setting (without any examples, only an instruction passed as the context), without actually updating model weights [37].

As for the capabilities of LLMs in the software engineering domain, a notable example is the code-specific model *Codex* from OpenAI [38]. It was shown to achieve good performance on a variety of software engineering tasks, including both code generation [38]–[41] and natural language generation [42]–[44]. Recently, two general-purpose LLMs were introduced by OpenAI — *GPT-3.5-turbo* (ChatGPT) and *GPT-4*. An overview of their capabilities can be found in the GPT-4 technical report [45] or in the survey from Liu et al. [46].

III. Commit Message Completion

Our first approach for improving the practical usefulness of existing solutions is moving from commit message generation to commit message completion. In this section, we formally define the task and describe the specifics of how it works with different model architectures.

Task definition. Commit message completion can be formulated as follows: given the prefix of a commit message and the *commit context*, generate a suitable continuation for the commit message. In practice, the length of this continuation may vary based on the quality and performance of the underlying approach and other factors. We follow the existing CMG works [8], [11], [15], [20] and use diff (*i.e.,* changes made in the commit) as the main commit context. Figure 2 presents the high-level overview for the commit message completion task with *commit contexts* that we consider. We focus on the language modeling approach to completion because it is widely adopted for natural language sequences [21], [34].

Input representation. In this study, we do not seek to utilize code structure and leave this to future research. We have two key observations to motivate our decision. Firstly, developers use version control systems not only for source code files but also for various related files (*e.g.*, configurations and READMEs), which do not necessarily follow a formal structure. Secondly, most of the existing approaches that utilize code structure only support Java [12], [18], [32], and it would require substantial effort to support more languages. To maintain the diversity necessary for the purposes of our study, we consider it essential to employ a dataset covering multiple programming languages and therefore use a plain textual representation of changes made in each commit — an output of the `git diff` command.

Encoder-decoder approach. Language modeling-based CMG approaches can be directly applied to the completion task. During training, we use the commit diff as the source sequence $\mathbf{x}$, and the ground truth commit message as the target sequence $\mathbf{y}$. During inference, the model takes a commit diff $\mathbf{x}$ and a prefix for a commit message $\mathbf{p_m}$. We pass $\mathbf{p_m}$ to the decoder since the output is expected to be its continuation. Optionally, we may include an additional *commit context* $\mathbf{c_m}$ (*e.g.*, containing this author's commit history). During training, we concatenate it with the ground truth commit message and build a target sequence $\mathbf{y_{c_m}} = (\mathbf{c_m}, [SEP], \mathbf{y})$, where $[SEP]$ is a special token. We compute and propagate loss only for the message $\mathbf{y}$. During inference, we concatenate it with the prefix $\mathbf{p_m}$ in the same manner.

LLM approach. In contrast with the CMG approaches, LLMs mostly follow the decoder-only architecture instead of the encoder-decoder. Also, we focus solely on evaluating the in-context learning abilities of LLMs, *i.e.,* we do not consider fine-tuning. Hence, several differences arise. The input for an LLM is a single sequence of tokens, often referred to as *prompt*. LLM outputs a continuation for the given prompt. There are many techniques for prompt engineering (*i.e.*, the process of choosing the best-performing prompt) [47], however, we focus on a simple zero-shot setting as a baseline. Specifically, we provide the model only with a simple instruction to complete the given commit message prefix based on commit diff and, optionally, additional *commit context*.

IV. Commit History & Diversity of Data

The second important improvement is utilizing the previous commit messages, as shown in Figure 2, which requires training the models on the appropriate data. In a recent overview, Tao et al. [15] highlighted that the majority of the available CMG datasets suffer from at least one of the following limitations: **(a)** only the Java language; **(b)** small scale of 20,000–100,000 commits; **(c)** limited information about each commit (hence, no way to trace back to the original commit on GitHub).

To mitigate these issues, Tao et al. [15] built a novel dataset called MCMD. However, we argue that there are two more important limitations in the existing CMG datasets: *significant tampering with the original commit history* and *restrictive data filtering*. The former undermines the validity of experiments with commit history and limits the possibility of taking into account individual characteristics of specific developers and projects, while the latter impedes the ability to learn potential conventions among a diverse range of commits and to evaluate on them. Let us now describe these limitations.

Tampering with the original commit history. To the best of our knowledge, at the time of publishing, our work is the first to explore the commit message history as an additional source of information for generating commit messages. We observe that preserving the original commit history was out of scope for the majority of existing published CMG datasets. This shortcoming manifests itself in one of the following ways: **(a)** preserving only a small fraction of the original set of commits due to strict filters [8], [20]; **(b)** performing train/validation/test split randomly, not by authors or by projects [8], [11], [12], [16], [20], [23], [32]; **(c)** downsampling (*i.e.,* selecting a subset) from the original set of commits randomly, without considering authors or projects [15], [16].

Restrictive data filtering. Even when it comes to individual commit messages, existing works usually apply restrictive filterings that relate both to the messages themselves and the diffs. Notice that we do not count as restrictive filtering those steps that aim to lower the number of automatically generated examples (*e.g.*, dropping merge or revert commits). Let us highlight the most common and important filters that researchers used in previous works. First, there are filters that relate to *commit messages*:

— *First Sentence*. Developers often use the first sentence of a commit message as a concise summary of the entire message [8]. Many CMG papers opt to extract the first sentence from commit messages as a target sequence.

— *Message Structure*. Commit messages from open-source projects vary drastically in terms of writing styles, and some filters are employed by CMG papers to restrict this variety. A notable example is the Verb-Direct Object (V-DO) filter, which only allows messages that start with a Verb followed by a Direct Object clause [8] (*e.g.*, *refactor code*, but not *minor refactoring*). Another option is filtering out commit messages that do not begin with one of the curated verbs [16].

— *Message Length*. This filter is usually targeting the number of tokens in commit messages.

Other filters target *commit diffs*:

— *Only Code*. Some studies only consider commits that only modify source code files (*i.e.*, `.java` for Java).

— *Diff Length*. This filter is usually targeting the number of tokens in diffs. Other variations include the number of changed files or chunks (changed lines grouped together).

We studied the corresponding papers and replication packages of existing CMG datasets and provide the resulting statistics on the usage of each filter in Table I. *First Sentence*, *Message Structure*, *Message Length*, and *Diff Length* are used in more than half of existing CMG datasets. The only dataset that does not employ any of these filters is the one from the work of Loyola et al. [9], however, it only contains commits from 12 specific projects. Also, we note that the dataset from the work of Jiang et al. [8] and its filtered version NNGen [20] are the most common for evaluation of CMG models [11], [13], [29], [30], and several subsequent works employ the same processing pipeline for their datasets [11], [23].

There is evidence for the restrictiveness of these filters. Jiang et al. [48] explored 1.6 million commit messages from top 1,000 Java projects, and their findings show that 53% of messages do not follow the Verb-Direct Object structure and 18% of the messages have more than one sentence. In the later work, Jiang et al. [8] employ the *Message Length* and *Diff Length* filters for 30 and 100 tokens, respectively. This filtered 1.8M commits into 75K commits (a reduction of 96%).

Drastic reductions used in previous research datasets impede the ability to study the history-based personalization of commit messages. In our work, we aim to bridge this gap by collecting a new large-scale, multilingual, history-aware, diverse dataset called *CommitChronicle*, and studying whether the history and filterings significantly influence the results.

V. The CommitChronicle Dataset

A. Data Collection

Choosing repositories. As our source of information, we chose GitHub, a large platform for hosting software projects. We used the GitHub Search tool [49] on January 25th, 2023 to select specific repositories for subsequent data mining. To filter only mature projects, we set the inclusion criteria based on the existing guidelines [50], similar to other works in SE research [51], [52]: 50+ stars, 10+ contributors, 1000+

TABLE I
STATISTICS ON RESTRICTIVE FILTERS IN EXISTING CMG DATASETS.

CMG Dataset	Commit message filters			Commit diff filters		
	First Sentence	**Message Structure**	**Message Length**	**Only Code**	**Diff Length (# tokens)**	**Diff Length (other)**
Loyola et al. [9]	–	–	–	–	–	–
Jiang et al. [8]	+	V-DO	$\leq$ 30 tokens	–	$\leq$ 100 tokens	–
NNGen. [20]	+	V-DO	$\leq$ 30 tokens	–	$\leq$ 100 tokens	–
PtrGNCMsg [11]	+	V-DO	$\leq$ 30 tokens	–	$\leq$ 100 tokens	–
CoDiSum [12]	–	–	$\leq$ 20 tokens	+	$\leq$ 200 tokens	–
CoReC [23]	+	V-DO	$\leq$ 30 tokens	–	$\leq$ 100 tokens	–
ATOM [32]	–	–	$\leq$ 20 tokens	+	–	$\leq$ 5 chunks
MCMD [15]	+	–	–	–	–	–
CommitBERT [16]	+	Verbs	–	+	–	$\leq$ 2 files
Total	6/9	5/9	6/9	3/9	5/9	2/9

commits, created at least two years ago, has a permissive license (Apache-2.0, MIT, BSD-3-Clause), not a fork. In total, we obtained 12.4k projects fitting our criteria.

Collection process. The data collection took place on February 9th, 2023. We used PyDriller [53] to collect commits. We collected all non-merge commits made after January 1st, 2017, opting to avoid earlier commits since they might be less relevant. In order to fit into reasonable resources for data processing, we also set the upper limit on the number of changed lines in a single commit to 10,000. In total, we obtained 27.4M commits.

B. Data Processing

Splitting by projects. To evaluate the models on the previously unseen projects and to avoid breaking the commit history, we split the data into the train, validation, and test sets by repositories with the 80%/10%/10% ratio adopted in previous work [15].

Filtering outliers. The main purpose of this stage is to drop examples that are both highly atypical and require a lot of time and memory to process. We calculated percentiles for the number of tokens (obtained via simple tokenization by whitespaces), number of characters, and number of modified files. We dropped examples out of the [5%, 95%] percentile range. We provide the exact percentile values in our online appendix [27]. This resulted in 21M commits, with 6.4M commits dropped (23.32% of commits from the initial step).

Commit message processing. Unlike most of the existing CMG datasets, we refrained from processing that would restrict the diversity of commit messages. Nevertheless, we adopted the best practices to filter out automatically generated or irrelevant commit messages, including messages with non-ASCII symbols [48], trivial messages [20], and merge and revert messages [8]. In addition, we identified and removed project-specific content from the messages, including URLs, emails, and references to issues or pull requests [54]. We provide all the regular expressions in our online appendix [27]. In total, this resulted in 19.2M commits, with 1.8M commits dropped (8.81% of commits from the previous step).

Commit diff processing. We store diffs as a list of file modifications, where each modification includes type (modifying file, creating file, deleting file, etc.), path to file before and after commit, and diff as obtained from Git. For diffs, we merge several consecutive whitespaces to save up disk space. Also, we drop commits with empty diffs. In total, this resulted in 18.6M commits, with 591K commits dropped (3.08% of commits from the previous step).

Deduplication. To ensure that our evaluation results are not inflated, we conducted exact hash deduplication of our dataset. Specifically, we group commits that share the same MD5 hash for their messages or diffs, and keep a single commit instance from each group. In total, we found and dropped 4.6M duplicate commits (24.59% of commits from the previous step). This resulted in 14.0M commits.

Dropping commits from overlapping authors. To prevent any overlap between commit authors in the training and evaluation sets, we removed all commits from authors who were present in either the validation or test set from the training set. Out of 488.5K authors in the training set, we identified 54.0K overlapping authors (11.07%). In total, this resulted in 10.8M commits, with 3.2M commits dropped (22.55% of commits from the previous step).

Dropping commits from bots. It is important to note that open-source projects frequently employ software bots to automate certain activities [55]–[58]. To further lower the number of automatically generated commits, we drop all commits from the authors that either are present in existing bot datasets [55], [58], [59] or have the suffix "bot" in their names [56]. Specifically, we identified 902 out of 603K authors as bots and dropped 165.8K commits (1.52% of commits from the previous step). In total, this resulted in 10.7M commits, which is the final number of commits in our dataset.

Name disambiguation. After all the processing, we replace the authors' names and emails with unique identifiers to prevent personal information disclosure. Since the same user might appear under several combinations of name and email (*e.g.*, work email and personal email), we employ a name disambiguation tool *gambit* [60] to obtain identifiers and merge authors together. *gambit* was shown to achieve near-perfect results for the authors from the Gnome GTK project. *gambit* calculates text similarity metrics for all pairs of authors: since processing all author pairs in our large dataset would be infeasible, we limited name disambiguation to the

TABLE II
NUMBER OF COMMITS IN THE OBTAINED COMMITCHRONICLE DATASET FOR EACH PROGRAMMING LANGUAGE. LANGUAGES ARE SORTED BY THE TOTAL NUMBER OF CORRESPONDING COMMITS.

Language	Train	Validation	Test	Total
Python	1,330,155	212,563	247,421	1,790,139
JavaScript	1,076,877	169,502	229,720	1,476,099
Java	952,162	200,035	204,862	1,357,059
TypeScript	936,697	177,258	198,272	1,312,227
C++	830,683	201,716	123,725	1,156,124
Go	672,045	133,954	134,699	940,698
C#	425,642	84,708	65,528	575,878
C	309,153	57,970	38,340	405,463
Rust	240,037	60,788	45,167	345,992
Ruby	181,916	39,912	33,433	255,261
PHP	178,556	32,293	36,618	247,467
Kotlin	154,276	29,781	28,021	212,078
Shell	117,927	13,902	27,019	158,848
Swift	101,274	28,227	8,938	138,439
Nix	2,526	86,022	8,108	96,656
Groovy	23,262	1,745	38,799	63,806
Dart	42,061	17,527	2,895	62,483
Elixir	41,562	3,380	5,874	50,816
Objective-C	32,517	1,294	7,708	41,519
Smalltalk	10,130	1,465	1,120	12,715
Total	7,659,458	1,554,042	1,486,267	10,699,767

authors committing to the same repository. Consequently, if some author committed to several repositories, they will have different identifiers for each repository. Due to this limitation, we performed two preceding steps related to commit authors before name disambiguation rather than after it.

C. Dataset Overview

General statistics. Table II presents the resulting number of commits in our dataset. Notice that we performed the splitting by repositories, which resulted in the uneven distribution in terms of commits, especially for low-resource languages (*e.g.*, Nix or Groovy). After all processing, we retained 10.7M of the initial 27.4M commits (38.98%), with the most restrictive stages being *Filtering outliers* and *Deduplication*, which are necessary for data cleaning. In terms of projects, we retained 11.9k of the initial 12.4k projects (95.97%). Following the MCMD dataset [15], we include not only diffs and messages but also rich metadata about each commit, including authors and timestamps necessary for experiments with the commit history, as well as repository names and commit hashes, which allow researchers to gather additional information if needed.

Filters and commit history statistics. In Section IV, we observed that many existing CMG datasets employ restrictive filters. *CommitChronicle* is large-scale and multilingual, hence, it is representative to assess the degree of restrictiveness for each filter that other researchers used previously. We implement the most frequent filters and calculate the number of affected commits for each of them. We provide the implementation for each filter in our online appendix [27]. Our findings show that *Diff Length* $\leq$ 100 *Tokens* and *Verb-Direct Object* are the most restrtictive filters: in our dataset, we would have excluded 84% and 64% commits, respectively, if we employed these filters. With *First Sentence*, 17% commits would be filtered out. Finally, *Message Length* $\leq$ 30 *Tokens* would only affect 7% commits, which suggests that commit messages tend to be concise in open source projects.

Another shortcoming of existing CMG datasets outlined in Section IV that makes them unsuitable for experiments with commit history is that they either do not provide the required metadata or tamper with the original commit history. In contrast, we provide the authors' identifiers and timestamps for all commits in our dataset. As for the commit histories, even after cleaning the data to ensure its quality, on average, we still retain 67% of commits in each history, which we consider enough for meaningful experiments.

VI. METHODOLOGY

We evaluate our two ideas for helping developers write high-quality commit messages: commit message completion instead of generation, as well as training and evaluating the models on the newly-collected history-aware and diverse *CommitChronicle* dataset. We conduct experiments with all four possible configurations shown in Figure 2. To thoroughly cover our two major ideas, we designed four research questions:

- **RQ A1**. How do state-of-the-art CMG approaches perform in the completion setting?
- **RQ A2**. How do LLMs perform in comparison with state-of-the-art CMG approaches?
- **RQ B1**. How does using commit message history as an additional input affect the models' quality?
- **RQ B2**. How do state-of-the-art CMG approaches perform with and without common data filtering steps?

In this section, we describe in detail the methodology for answering these research questions.

Models. Since our dataset includes many programming languages rather than focusing on a single one, this makes it infeasible to adapt the approaches that utilize code structure. Also, retrieval approaches are not directly applicable to the completion task. Therefore, we focus on language modeling or hybrid approaches that do not require structural information.

Specifically, we consider approaches that were shown to achieve the best performance in a recent CMG study [19]. We also expand the list with a recent model that was shown to achieve superior performance on a variety of tasks related to commit diffs, as well as a powerful LLM.

CodeT5 [24] is a variation of the sequence-to-sequence language model T5 [61] that was pretrained on a large amount of source code with source code-specific pretraining objectives. It was shown to achieve good performance on a variety of downstream tasks, including CMG [19].

RACE [19] is a hybrid CMG approach that utilizes similar commit messages and commit diffs to improve the quality of a language model. RACE with CodeT5 as a backbone is state of the art in CMG at the time of writing. We observed that the retrieval implementation in the RACE replication package is too demanding for the scale of our dataset both in terms of memory and time complexity. Hence, we reimplemented RACE with the retrieval powered by the open-source Approximate Nearest Neighbors (ANN) tool *annoy* [62].

CodeReviewer [25] is a variation of CodeT5 that was further pretrained on a large-scale dataset of diffs and code review comments from GitHub. While code review comments and commit messages serve different purposes, we still consider CodeReviewer relevant to the CMG task. In particular, its pretraining includes objectives for understanding commit diffs, which has the potential to bring value for the CMG task.

GPT-3.5-turbo is a recent general-purpose LLM from OpenAI. It is officially recommended to be used instead of Codex, as Codex has been deprecated as of March 2023 [63]. We only aim to obtain a reasonable baseline of possible CMG performance of LLMs, hence, we leave the experiments with the more sophisticated GPT-4 model to future research.

Training and evaluation setting. For all CMG models (*i.e.,* all models except for GPT-3.5-turbo), we choose the *base* configuration, since it is the only configuration available for CodeReviewer, and it was shown to be superior over *small* configuration [19]. For all models, we set the maximum source length and target length to 512 tokens.

We use the mixed precision for CMG models to accelerate both training and evaluation. We follow the hyperparameters setting from the most recent CMG study [19]. Specifically, we use the AdamW optimizer with 2×10^{-5} peak learning rate, no weight decay, and a linear warmup strategy with 100 warm-up steps. The training is conducted either on a single NVIDIA TITAN RTX GPU or on 4 NVIDIA T4 GPUs via the Distributed Data Parallel (DDP) framework. Effective batch size is set to 32 for all runs, with gradient accumulation due to the limited GPU memory when needed. Due to the large-scale nature of our dataset, we train all the models for 1 epoch only, which makes 7.6M examples and around 3B tokens. During the evaluation, we use beam search with width 5 as the decoding strategy, set the maximum number of generated tokens to 15, and allow early stopping when a special end-of-sequence token is produced.

For GPT-3.5-turbo, we access the model through the official OpenAI API. OpenAI API provides several configurable parameters for completion endpoints. We set the temperature to 0.8 and top-p to 0.95, as it was done in previous works [38], [41]. To match our setting for the CMG approaches, we set the maximum generation length to 15 tokens and truncate the diffs to 512 tokens when constructing prompts.

Models' input. We mimic a real-world completion scenario by passing the first $X\%$ of characters of each commit message into the context. We experiment with several values of X — 0% (generation setting), 25% and 50% (completion settings).

For GPT-3.5-turbo, we instruct it with zero-shot prompting. Note that it is a chat model, so the expected data format differs from completion models. We rely on official guidelines to properly define the input in the required format. We provide all the prompts in our online appendix [27].

All the CMG models that we consider utilize Byte-Pair Encoding (BPE) [64] subword tokenization algorithm. Hence, for completion, we face an issue with the tokenization of the last incomplete word in the prefix outlined by Popov et al. [65]. To mitigate it, we remove the last incomplete word from the model input and restrict the beam search to produce outputs consistent with the removed word part. For GPT-3.5-turbo in the completion setting, we explicitly instruct it to continue given prefixes rather than generate messages from scratch.

We also experiment with commit message history as additional context, following the approach we described in Section III. For CMG models, we use as many previous commit messages as fit into the context when concatenated with the ground truth message, separating historical messages with a special token $[SEP]$. For GPT-3.5-turbo, we opt for a simpler setting due to its availability through paid API and extend the prompts only with a single previous message from the commit histories of the respective authors within the same repository. Hence, we probe a baseline for what GPT-3.5-turbo may achieve with commit message history.

Evaluation metrics. We employ three metrics commonly used to evaluate the quality of generated commit messages:

- *B-Norm* is a version of BLEU [28] that was shown to be the most in line with human judgment on the quality of commit messages by Tao et al. [15]. BLEU is a metric based on the precision in terms of overlapping n-grams (*i.e.,* contiguous sequences of *n* words) between generated and reference sequences. It is used in most of prior commit message generation works [8], [15], [20].
- *Edit Similarity* (*EdSim*) is a metric based on the Levenshtein distance [66] between the predicted and reference texts. It is suitable for evaluating completion systems because users can accept slightly wrong suggestions as long as not too many edits are required [22].
- *ExactMatch* (*EM*) measures the percentage of predicted sequences that exactly match the reference text.

We not only report the metrics between full predictions and targets but also investigate how the quality evolves with the number of generated tokens. In addition to the metrics for full sequences, we calculate metric values between prefixes of predictions and targets. For *ExactMatch*, we report the values for prefixes of 1 and 2 tokens, since perfect predictions quickly deteriorate to near-zero values. For *B-Norm*, we report metrics for prefixes of 4–10 tokens, since the implementation we use expects 4-grams to contribute to the final score. For *EdSim*, we report metrics for 1–10 tokens.

Data. We use the *CommitChronicle* dataset described in detail in Section V. However, we observe that generating predictions for the whole test set with 1.5 million commits would require an implausible amount of time. We obtain a subsample CMG_{test} as follows: **(a)** we exclude the repositories with the number of commits more than 95% percentile (4,161 commits); **(b)** we downsample repositories for frequent languages to 17 repositories, to cap the number of commits at around 20K per language. We do not downsample less represented languages from Table II, and keep a reasonable trade-off between the data quantity and diversity. In total, CMG_{test} contains $204,336$ commits.

Moreover, since we focus on LLMs available through paid API, we select a smaller yet subsample for all related experiments to keep the costs reasonable. Specifically, we

TABLE III

RESULTS FOR CMG APPROACHES ON CMG_{test}. *EdSim* STANDS FOR *Edit Similarity*, *EM* STANDS FOR *ExactMatch*. *CoRev* IS THE CODEREVIEWER MODEL [25]. PERCENTAGES REPRESENT THE LENGTH OF THE PROVIDED MESSAGE PREFIX, WITH 0% BEING THE GENERATION TASK.

	Approach	B-Norm	EdSim	EM@1	EM@2	№
0% (Gen.)	CodeT5, history	16.80	30.91	17.68	4.27	1
	RACE, history	**16.91**	**31.15**	**17.95**	4.36	2
	CoRev, history	16.78	30.74	17.83	**4.38**	3
	CodeT5	15.12	28.71	10.90	3.03	4
	RACE	**15.32**	**29.02**	**11.37**	**3.07**	5
	CoRev	15.15	28.76	10.87	3.05	6
25% (Compl.)	CodeT5, history	21.94	33.31	44.98	13.10	7
	RACE, history	**22.16**	**33.78**	45.36	**13.40**	8
	CoRev, history	21.84	32.90	**45.58**	13.28	9
	CodeT5	17.91	30.54	45.35	12.92	10
	RACE	**18.38**	30.91	**46.62**	**13.45**	11
	CoRev	18.10	**30.93**	46.05	13.35	12
50% (Compl.)	CodeT5, history	26.90	33.95	47.45	12.75	13
	RACE, history	**27.28**	**34.69**	47.84	**13.26**	14
	CoRev, history	26.94	33.76	**48.10**	12.90	15
	CodeT5	24.13	32.74	49.94	14.03	16
	RACE	**24.74**	**33.22**	50.68	14.38	17
	CoRev	24.35	33.20	**50.90**	**14.59**	18

obtain LLM_{test} by randomly selecting 10 authors with 10–50 commits for each programming language from CMG_{test}. In total, LLM_{test} contains $4,025$ commits. Note that many previous works used comparable or even smaller datasets when experimenting with LLMs [38], [41], [43], [44].

VII. RESULTS & DISCUSSION

Let us now describe the results of our experiments. We use paired bootstrap resampling [67] with 99% confidence level to test statistical significance across different models and settings, *i.e.,* to compare a pair of models, we randomly sample with replacement, generating a sample of the same size as the test set, and compute metrics for both models on this new sample; we repeat the process $1,000$ times and declare a winner only if it outperforms the other model in 99% of cases. Due to the lack of space, we only share several plots with metrics between prefixes related to particularly interesting findings. We provide all the remaining plots in our online appendix [27].

A. Commit Message Completion

RQ A1: How do state-of-the-art CMG approaches perform in a completion setting? We present the metrics for the CMG approaches in Table III. First, we observe that the values of metrics tend to grow as we increase the context ratio (*i.e.*, go from generation to completion), especially in *ExactMatch* and *B-Norm* metrics. For example, for CodeT5, *ExactMatch@1* grows from 10.90 (line 4, 0%) to 45.35 (line 10, 25%), to 49.94 (line 16, 50%). We confirm that improvements for all models and metrics are statistically significant. Hence, to some extent, we confirm the previous finding [34] — completion is a simpler task than generation and might be a good choice to employ existing models to bring value to actual users.

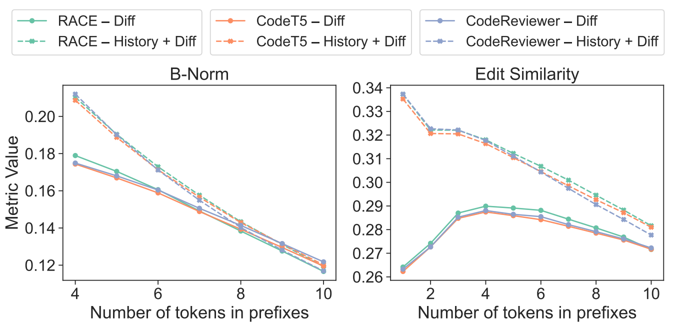


Fig. 3. Metrics between prefixes on CMG_{test} for 0% context ratio (generation setting) for CMG approaches. Dashed lines correspond to models with history.

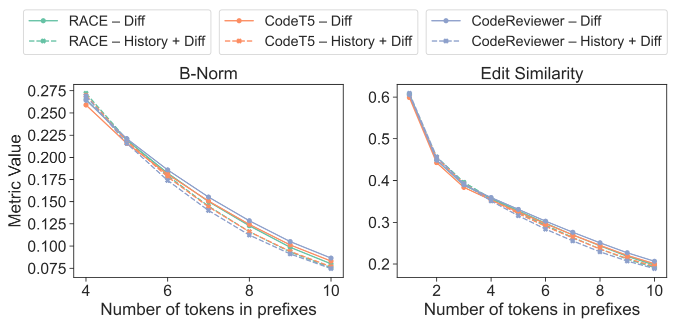


Fig. 4. Metrics between prefixes on CMG_{test} for 25% context ratio (completion setting) for CMG approaches. Dashed lines correspond to models with history.

Next, we observe that there is but a small difference in absolute metrics values between state-of-the-art CMG approaches, and their ranking is not consistent across different metrics and different context ratios. Additionally, we observe that plots for metrics between prefixes for all the models exhibit very similar patterns — the models are almost indistinguishable. We share the plots for the 0% context ratio setting in Figure 3 and the plots for the 25% context ratio setting in Figure 4.

We confirm that the superiority of RACE over both CodeT5 and CodeReviewer is statistically significant in all context ratios. In the 0% context ratio setting, the difference between CodeT5 and CodeReviewer is not statistically significant, however, CodeReviewer is confirmed to be superior in both completion settings. This outcome is not completely surprising, since all the models are based on CodeT5, however, previous works suggest significantly larger gaps in performance. For RACE, our setup is different from the original in two aspects: we include more diverse commits in the dataset and we switch from the exact retrieval implementation to the approximate one. As for CodeReviewer, perhaps, the fact that it was intended for the code review tasks plays a larger role, since code review data can be different from commit messages.

Finally, from Figure 4 we observe that all the metrics values in the completion setup decrease with the prediction length, which is consistent with the work on comment completion [34]. However, for generation, from Figure 3 we note that *Edit Similarity* for short sequences is worse than for the se-

TABLE IV
RESULTS FOR CMG APPROACHES AND GPT-3.5-TURBO ON LLM_{test}. ALL THE RESULTS ARE PRESENTED FOR THE 25% CONTEXT RATIO (COMPLETION SETTING). *EdSim* STANDS FOR *Edit Similarity*, *EM* STANDS FOR *ExactMatch*. *CoRev* IS THE CODEREVIEWER MODEL [25].

Approach	B-Norm	EdSim	EM@1	EM@2	№
CodeT5, history	21.11	32.31	43.68	12.45	1
RACE, history	21.14	32.35	44.92	12.93	2
CoRev, history	**21.35**	**32.68**	**45.69**	**13.15**	3
CodeT5	17.16	30.02	45.19	12.85	4
RACE	**17.54**	30.13	**46.68**	**13.33**	5
CoRev	17.34	**30.45**	45.96	13.03	6
GPT-3.5-turbo, history	**13.24**	**27.83**	**34.34**	**10.09**	7
GPT-3.5-turbo	11.48	26.35	21.84	5.99	8

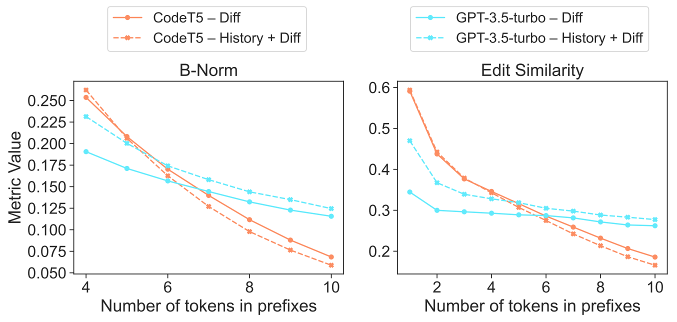


Fig. 5. Metrics between prefixes on LLM_{test} for 25% context ratio (completion setting) for CodeT5 and GPT-3.5-turbo. Dashed lines correspond to models with history.

quences of average length. We investigate the examples where *Edit Similarity* for sequences of 4 tokens is higher than *Edit Similarity* for sequences of 2 tokens. We find that many cases are due to conventions, which suggest special prefixes for commit messages, *e.g.,* Conventional Commits [68]. Consider this example: the ground truth commit message is `feat: add hideSampleTab option`, and the corresponding prediction is `Add hideSampleTab option`. CMG approaches can fail to correctly determine the convention for the particular case. This finding highlights the variety of writing styles and conventions for commit messages that exist in the wild. Since in the completion setup all the metrics decrease, this problem may be less relevant to completion.

Summary of RQ A1. *The completion task is easier than generation. All CMG approaches exhibit similar patterns, but RACE is better than both CodeT5 and CodeReviewer. For completion, performance consistently decreases as the number of tokens to predict grows. For generation, there is an additional challenge of correctly generating the beginning of the sequence.*

RQ A2: How do LLMs perform in comparison with state-of-the-art CMG approaches? To answer this research question, we obtain predictions from GPT-3.5-turbo on the LLM_{test} dataset. Since LLM_{test} is a subset of CMG_{test}, we also recalculate the metrics for all CMG approaches using only predictions for LLM_{test}. We present the results in Table IV. In Figure 5, we also present plots for metrics between prefixes on LLM_{test}. Similar to CMG_{test}, different CMG approaches exhibit very similar patterns, so we only include metrics for CodeT5 for clarity. Due to the lack of space, we only include the results for the 25% context ratio. We note that our findings mostly hold true for the remaining settings as well. We provide the rest of the results in our online appendix [27].

While the results of the CMG approaches on LLM_{test} are slightly different from the results on a larger dataset CMG_{test}, they remain of the same magnitude, and most patterns still hold true. Hence, we consider LLM_{test} a reliable dataset to estimate LLM performance. We observe that the metrics values between full predictions and targets are lower for GPT-3.5-turbo (line 8) than for the CMG approaches (lines 4–6). We confirm that the difference is statistically significant.

Figure 5 shows that while GPT-3.5-turbo is worse than CodeT5 for short sequences, it becomes better as the number of tokens to predict grows. By empirically investigating predictions, we note that GPT-3.5-turbo often tries to produce detailed messages and completely exhausts the maximum tokens restriction, while CMG models often stop early. In the 25% context ratio setting, the median number of tokens for GPT-3.5-turbo and for all CMG models is 9 and 4, respectively. Additionally, we observe that GPT-3.5-turbo may disregard the given prefixes and generate novel messages instead. With CMG approaches, we explicitly restrict the beam search procedure to consider only hypotheses starting from the given prefixes, which is not possible for the LLM available through the API. This might be the underlying reason for the CMG approaches being especially superior in terms of metrics between short prefixes and *ExactMatch*.

Summary of RQ A2. *Generally, and especially for short sequences, GPT-3.5-turbo is worse than CMG approaches. However, it may be a better choice for generating long and detailed messages. Moreover, we note that our LLM setting is very simple, and further investigation is required to uncover the full potential of LLMs in commit message generation and completion tasks.*

B. History and Diversity of Data

RQ B1: How does using commit message history as an additional input affect the models' quality? From Table III, we observe that *B-Norm* and *Edit Similarity* metrics across all the models and settings increase when adding history. In the generation setting, *ExactMatch* is drastically better for the models with history (lines 1–3) compared to the models without history (lines 4–6). However, for completion, history slightly distracts the models (lines 7–9 with history and lines 10–12 without history, 25% context ratio). We confirm that the impact of history in terms of *B-Norm* and *Edit Similarity* is statistically significant, as well as for *ExactMatch* in the generation setting. However, not all the models exhibit statistically significant difference in *ExactMatch* in completion settings.

From Figure 3 with the plots with metrics between prefixes in the generation setting, we observe that history improves the results on short sequences. Hence, commit message history might be helpful to determine suitable conventions for the current author and repository when generating messages from scratch. However, plots with metrics between prefixes in the completion setting from Figure 4 show that adding the history barely impacts the completion of a given message prefix: the plots for the models with and without history follow similar patterns, and the models with history are even worse in some cases. This might indicate that the beginning of the commit message is extra challenging to generate correctly. Since the beginning of the commit message is given in the completion setting, the commit message history brings little value.

Apart from the CMG approaches, we also experiment with providing GPT-3.5-turbo a previous message from the commit message history. We observe that GPT-3.5-turbo benefits from history (lines 7-8 from Table IV) and we confirm that the difference is statistically significant.

Summary of RQ B1. *Our approach of integrating commit message history into CMG approaches turned out to be useful for generation, but the results for completion are conflicting. In contrast, GPT-3.5-turbo benefits from the history both in the completion and generation setting. Overall, commit message history holds potential, but more exploration is necessary.*

RQ B2: How do state-of-the-art CMG approaches perform with and without common data filtering steps? To answer this research question, we study the most frequent filters, namely, *First Sentence*, *Verb-Direct Object*, and *Diff Length* ≤ 100 *tokens*. We do not consider *Message Length* ≤ 30 *tokens*, because all the models are allowed to generate only up to 15 tokens, so naturally the metrics for the subset where all the targets are longer would be low.

In our CMG_{test} dataset, $10,385$ examples (5.08%) fit all the filters and $22,332$ (10.93%) examples fit neither of the filters. In Table V, we present the metrics for the Filtered subset, the Out-of-Filters subset of $10,385$ examples (fitting *no* filters), and a random subset of $10,385$ examples. Due to the lack of space, we only include the results for the 25% context ratio. We note that our findings mostly hold true for the remaining settings as well. We provide the rest of the results in our online appendix [27].

We observe from Table V that all CMG approaches achieve higher results on the Filtered subset (lines 4–6) than on the Random subset (lines 10–12), and the results on the Out-of-Filters subset (lines 16–18) are by far the lowest (except for *ExactMatch*). The difference is statistically significant for all the models, with an exception for *ExactMatch* for some cases. Therefore, using the Filtered subset may not provide a reliable estimate of the quality on a diverse set of commits. Moreover, generation or completion for Out-of-Filters commits is more challenging, and current approaches seem to struggle with this task. Note that the data in the Out-of-Filters subset was subject to cleaning as described in Section V.

TABLE V
Results for CMG approaches on filtered, random, and out-of-filters subsets of CMG_{test} with $10,385$ examples. All the results are presented for the 25% context ratio (completion setting). *EdSim* stands for *Edit Similarity*, *EM* stands for *ExactMatch*. *CoRev* is the CodeReviewer model [25].

	Approach	B-Norm	EdSim	EM@1	EM@2	№
Filtered	CodeT5, history	29.21	39.36	51.55	16.29	1
	RACE, history	**29.39**	**40.25**	**52.35**	**17.00**	2
	CoRev, history	29.22	39.35	51.99	16.89	3
	CodeT5	22.77	35.11	48.51	15.55	4
	RACE	**23.48**	**35.85**	**49.91**	**16.28**	5
	CoRev	22.88	35.56	49.04	15.67	6
Random	CodeT5, history	21.74	33.24	44.26	13.43	7
	RACE, history	**22.16**	**33.81**	**45.12**	**13.72**	8
	CoRev, history	21.71	32.92	45.11	13.42	9
	CodeT5	17.66	30.48	44.88	12.93	10
	RACE	**18.28**	**30.95**	**46.24**	**14.05**	11
	CoRev	18.04	30.94	45.51	13.82	12
Out-of-filters	CodeT5, history	**5.61**	**16.00**	44.55	**10.52**	13
	RACE, history	5.52	15.54	44.48	10.02	14
	CoRev, history	5.41	15.42	**45.38**	10.40	15
	CodeT5	6.21	16.44	47.41	10.98	16
	RACE	6.16	16.21	47.68	11.47	17
	CoRev	**6.37**	**16.71**	**48.79**	**11.64**	18

We also check whether any of our previous findings about commit message history differ when we consider only Filtered or only Out-of-Filters subsets. In Table V, when using commit message history together with diffs on the Filtered subset (lines 1–3), CMG approaches perform better than when only using diffs (lines 4–6). These results are statistically significant. Moreover, we observe that the difference between the models with and without history tends to be higher for the Filtered subset than it is for the Random. For example, consider RACE: the increase in *B-Norm* on the Filtered subset is 25.17% (from 23.48 to 29.39), while on the Random subset, it is 21.23% (from 18.28 to 22.16). In contrast, on the Out-of-filters subset, additional history input decreases the performance of the CMG approaches. In completion settings, these results are statistically significant. These results may indicate that choosing the correct conventions is not the only challenge for commits that do not fit the common filters or that our method of integrating commit message history into CMG approaches is not tailored for this specific case.

Summary of RQ B2. *For the Out-of-Filters subset, the inclusion of history does not improve the results, which requires additional exploration. The average results of CMG approaches are different from both Filtered and Out-of-Filters subsets — Filtered results are better, while Out-of-Filters results are worse. Hence, the usage of restrictive filtering makes the evaluation results less representative.*

VIII. Related work

A question of the practical applicability of existing commit message generation approaches was raised in the recent study [14]. In several studies, over 50% of generated messages were rated as low-quality by human experts [8], [14], [20]. As a way to overcome this issue, Wang et al. [14] propose a quality assurance framework *QACom* that determines whether a given message is semantically relevant for a given diff. It can be used on top of any CMG approach to avoid showing low-quality predictions to users.

In our work, we propose to explore another option — repurposing commit message generation approaches for the completion task. Researchers have already tackled the completion of code comments [34], [69]. Mastropaolo et al. [34] explored the performance of a simple n-gram model that uses only the prefix of a comment typed by the developer as an input, as well as a Text-To-Text Transfer Transformer (T5) [61] that utilizes both the prefix and the related code context. Commit messages are similar to code comments in the sense that both are natural language artifacts for software maintenance and comprehension. However, both domains have their unique challenges. To the best of our knowledge, there is no published work on the completion of commit messages.

IX. Threats to Validity

Internal validity. We identified and addressed these threats:

Data quality. Since we collect a large number of commits from open-source repositories, their quality is a potential threat to validity. To mitigate this threat, we followed best practices from previous work to clean the data (*e.g.,* deduplicate and remove messages from bots) and extensively described our processing pipeline.

Hyperparameters. Since we consider several approaches and settings and train them on a large-scale dataset, it is infeasible to tune optimal hyperparameters for each individual approach. To mitigate this threat, we employed hyperparameters setting from a previous study by Shi et al. [19].

Implementation Errors. The presence of bugs in the source code implemented for this study poses a potential threat to validity. To mitigate this threat, we thoroughly tested the code and followed software engineering best practices to ensure that it was written in a clear and organized manner. Despite our efforts, there is still a possibility of undetected errors being present in the implementation.

External validity. We identified the following threats:

Model selection. We included several CMG approaches and a recent general-purpose LLM in our experiments. Still, our findings may not generalize to the models that were not considered in this study.

Dataset. Our findings may not transfer to different kinds of commits that were not present in our dataset, for instance, commits in other programming languages (beyond the 20 included in our dataset) or commits from proprietary codebases.

Even though these threats are important to note, we believe that they do not invalidate the main contributions of this work and the important novel ideas that we evaluated.

Verifiability. We provide all the necessary details about our study to help others replicate. We released publicly the data [26] and all the code for our models and experiments [27].

X. Future Work

Our future work aims to expand upon the two ideas presented in this study. In particular, we only considered a single way of integrating commit message history, and other approaches may result in further improvements. Moreover, our approach implies extending models' inputs with previous commit messages. All of the models in our experiments follow the Transformer architecture [36], whose time complexity scales quadratically with the number of tokens in inputs, so longer contexts might harm the performance. Hence, a promising direction might be to develop a separate model for producing efficient representations of developers' writing style or project conventions based on commit message history. Likewise, we only investigated a single zero-shot setting for a single LLM, and other LLMs or more advanced prompt engineering techniques are worth exploring.

Another important research direction is semantic evaluation. We followed best practices by using the B-Norm metric [15], however, all the metrics we employed are based on the overlap of words or characters between the generated and the reference messages, which may not fully reflect the semantic quality aspects, such as adequacy or usefulness. Human evaluation might uncover new insights for all the RQs addressed in our work. Alternatively, it could be intriguing to investigate the applicability of the recent automatic semantic metrics, including using LLMs as evaluators [70], for CMG task.

Finally, expanding the scope to diverse commits brings novel challenges. Firstly, there is a possible concern of data quality, especially regarding the commit messages. Considering that the notion of commit message quality gained attention in several recent studies [2], [3], additional filtering of our dataset might be required. Secondly, we believe that conducting a deeper analysis of the commits that current CMG approaches struggle with could benefit the field.

XI. Conclusions

This work sheds light on the potential of two novel ideas for personalized commit message generation, namely, *focusing on the completion task* and *integrating commit message history as context*. We conduct experiments with several CMG approaches, including previously proposed models from the literature, and a recent LLM GPT-3.5-turbo. Our results suggest that both ideas show promise when implemented independently, however, yield questionable improvements when implemented together. Furthermore, we highlight that several data filtering steps employed in previous works are overly restrictive and exclude many real-world examples. Based on our findings, relying solely on the results obtained on filtered commits might not provide a reliable estimate of the overall performance. In contrast, commits that fall outside of the common filters present new challenges to existing CMG approaches. Finally, we show that the overall quality of GPT-3.5-turbo in the zero-shot setting is inferior to existing state-of-the-art CMG approaches, but it has the potential for generating detailed commit messages.

References

[1] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "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*, 2012, pp. 1–11.

[2] J. Li and I. Ahmed, "Commit message matters: Investigating impact and evolution of commit message quality," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 806–817.

[3] Y. Tian, Y. Zhang, K.-J. Stol, L. Jiang, and H. Liu, "What makes a good commit message?" in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2389–2401.

[4] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 422–431.

[5] R. P. L. Buse and W. R. Weimer, "Automatically documenting program changes," in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software eEngineering*, 2010, pp. 33–42.

[6] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 275–284.

[7] J. Shen, X. Sun, B. Li, H. Yang, and J. Hu, "On automatic summarization of what and why information in source code changes," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, 2016, pp. 103–112.

[8] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 135–146.

[9] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "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)*, 2017, pp. 287–292.

[10] P. Loyola, E. Marrese-Taylor, J. Balazs, Y. Matsuo, and F. Satoh, "Content aware source code change description generation," in *Proceedings of the 11th International Conference on Natural Language Generation*, 2018, pp. 119–128.

[11] Q. Liu, Z. Liu, H. Zhu, H. Fan, B. Du, and Y. Qian, "Generating commit messages from diffs using pointer-generator network," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 299–309.

[12] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, "Commit message generation for source code changes," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, 2019, pp. 3975–3981.

[13] L. Y. Nie, C. Gao, Z. Zhong, W. Lam, Y. Liu, and Z. Xu, "CoreGen: Contextualized code representation learning for commit message generation," *Neurocomputing*, vol. 459, pp. 97–107, 2021.

[14] B. Wang, M. Yan, Z. Liu, L. Xu, X. Xia, X. Zhang, and D. Yang, "Quality assurance for automated commit message generation," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 260–271.

[15] W. Tao, Y. Wang, E. Shi, L. Du, S. Han, H. Zhang, D. Zhang, and W. Zhang, "On the evaluation of commit message generation models: An experimental study," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 126–136.

[16] T. H. Jung, "CommitBERT: Commit message generation using pre-trained programming language model," in *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, 2021, pp. 26–33.

[17] J. Bai, L. Zhou, A. Blanco, S. Liu, F. Wei, M. Zhou, and Z. Li, "Jointly learning to repair code and generate commit message," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 9784–9795.

[18] J. Dong, Y. Lou, Q. Zhu, Z. Sun, Z. Li, W. Zhang, and D. Hao, "FIRA: fine-grained graph-based code change representation for automated commit message generation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 970–981.

[19] E. Shi, Y. Wang, W. Tao, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "RACE: Retrieval-augmented commit message generation," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, 2022, pp. 5520–5530.

[20] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: How far are we?" in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 373–384.

[21] M. X. Chen, B. N. Lee, G. Bansal, Y. Cao, S. Zhang, J. Lu, J. Tsay, Y. Wang, A. M. Dai, Z. Chen *et al.*, "Gmail smart compose: Real-time assisted writing," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 2287–2295.

[22] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[23] H. Wang, X. Xia, D. Lo, Q. He, X. Wang, and J. Grundy, "Context-aware retrieval-based deep commit message generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, pp. 1–30, 2021.

[24] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "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*, 2021, pp. 8696–8708.

[25] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu *et al.*, "Automating code review activities by large-scale pre-training," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1035–1047.

[26] A. Eliseeva, Y. Sokolov, E. Bogomolov, Y. Golubev, D. Dig, and T. Bryksin. (2023) The CommitChronicle dataset. [Online]. Available: https://doi.org/10.5281/zenodo.8189044

[27] ——. (2023) The source code and supplementary materials. [Online]. Available: https://github.com/JetBrains-Research/commit_message_generation

[28] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[29] K. Etemadi and M. Monperrus, "On the relevance of cross-project learning with nearest neighbours for commit message generation," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 470–475.

[30] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "CC2Vec: Distributed representations of code changes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 518–529.

[31] Y. Huang, N. Jia, H.-J. Zhou, X.-P. Chen, Z.-B. Zheng, and M.-D. Tang, "Learning human-written commit messages to document code changes," *Journal of Computer Science and Technology*, vol. 35, no. 6, pp. 1258–1277, 2020.

[32] S. Liu, C. Gao, S. Chen, L. Y. Nie, and Y. Liu, "ATOM: Commit message generation based on abstract syntax tree and hybrid ranking," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1800–1817, 2020.

[33] V. Bibaev, A. Kalina, V. Lomshakov, Y. Golubev, A. Bezzubov, N. Povarov, and T. Bryksin, "All you need is logs: Improving code completion by learning from anonymous IDE usage logs," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1269–1279.

[34] A. Mastropaolo, E. Aghajani, L. Pascarella, and G. Bavota, "An empirical study on code comment completion," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 159–170.

[35] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. V. Franco, and M. Allamanis, "Fast and memory-efficient neural code completion," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 329–340.

[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[37] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," vol. 33, 2020, pp. 1877–1901.

[38] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[39] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 919–931.

[40] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring LLM-based general bug reproduction," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2312–2323.

[41] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1482–1494.

[42] J. Y. Khan and G. Uddin, "Automatic code documentation generation using GPT-3," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–6.

[43] T. Ahmed and P. Devanbu, "Few-shot training llms for project-specific code-summarization," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[44] T. Ahmed, S. Ghosh, C. Bansal, T. Zimmermann, X. Zhang, and S. Rajmohan, "Recommending root-cause and mitigation steps for cloud incidents using large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, p. 1737–1749.

[45] OpenAI, "GPT-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[46] Y. Liu, T. Han, S. Ma, J. Zhang, Y. Yang, J. Tian, H. He, A. Li, M. He, Z. Liu *et al.*, "Summary of ChatGPT/GPT-4 research and perspective towards the future of large language models," *arXiv preprint arXiv:2304.01852*, 2023.

[47] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "ChatGPT prompt patterns for improving code quality, refactoring, requirements elicitation, and software design," *arXiv preprint arXiv:2303.07839*, 2023.

[48] S. Jiang and C. McMillan, "Towards automatic generation of short summaries of commits," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 320–323.

[49] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in GitHub for MSR studies," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 560–564.

[50] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining GitHub," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 92–101.

[51] T. Wang, Y. Golubev, O. Smirnov, J. Li, T. Bryksin, and I. Ahmed, "PyNose: A test smell detector for python," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 593–605.

[52] K. Grotov, S. Titov, V. Sotnikov, Y. Golubev, and T. Bryksin, "A large-scale comparison of Python code in Jupyter notebooks and scripts," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 353–364.

[53] D. Spadini, M. Aniche, and A. Bacchelli, "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*, 2018, pp. 908–911.

[54] Z. Liu, X. Xia, C. Treude, D. Lo, and S. Li, "Automatic generation of pull request descriptions," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 176–188.

[55] M. Golzadeh, A. Decan, D. Legay, and T. Mens, "A ground-truth dataset and classification model for detecting bots in GitHub issue and PR comments," *Journal of Systems and Software*, vol. 175, p. 110911, 2021.

[56] M. Golzadeh, A. Decan, and N. Chidambaram, "On the accuracy of bot detection techniques," in *Proceedings of the Fourth International Workshop on Bots in Software Engineering*, 2022, pp. 1–5.

[57] T. Dey, S. Mousavi, E. Ponce, T. Fry, B. Vasilescu, A. Filippova, and A. Mockus, "Detecting and characterizing bots that commit code," in *Proceedings of the 17th international conference on mining software repositories*, 2020, pp. 209–219.

[58] A. Abdellatif, M. Wessel, I. Steinmacher, M. A. Gerosa, and E. Shihab, "BotHunter: An approach to detect software bots in GitHub," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 6–17.

[59] M. Golzadeh, A. Decan, and T. Mens, "Evaluating a bot detection model on git commit messages," *arXiv preprint arXiv:2103.11779*, 2021.

[60] C. Gote and C. Zingg, "Gambit–an open source name disambiguation tool for version control systems," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 80–84.

[61] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 1, pp. 5485–5551, 2020.

[62] Spotify. (2023) Annoy: Approximate Nearest Neighbors in C++/Python optimized for memory usage and loading/saving to disk. [Online]. Available: https://github.com/spotify/annoy

[63] OpenAI. (2023) Code completion (deprecaated). [Online]. Available: https://platform.openai.com/docs/guides/code

[64] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 1715–1725.

[65] A. Popov, D. Orekhov, D. Litvinov, N. Korolev, and G. Morgachev, "Time-efficient code completion model for the R programming language," in *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, 2021, pp. 34–39.

[66] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet physics. Doklady*, vol. 10, pp. 707–710, 1965.

[67] P. Koehn, "Statistical significance tests for machine translation evaluation," in *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, 2004, pp. 388–395.

[68] C. Commits. (2023) A specification for adding human and machine readable meaning to commit messages. [Online]. Available: https://www.conventionalcommits.org/en/v1.0.0/

[69] A. Ciurumelea, S. Proksch, and H. C. Gall, "Suggesting comment completions for Python using neural language models," *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 456–467, 2020.

[70] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing *et al.*, "Judging LLM-as-a-judge with MT-Bench and chatbot arena," *arXiv preprint arXiv:2306.05685*, 2023.