CommitBench: A Benchmark for Commit Message Generation

Maxmilian Schall
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
Maximilian.Schall@hpi.de

Tamara Czinczoll
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
Tamara.Czinczoll@hpi.de

Gerard de Melo
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
Gerard.deMelo@hpi.de

***Abstract*—Writing commit messages is a tedious daily task for many software developers, and often remains neglected. Automating this task has the potential to save time while ensuring that messages are informative. A high-quality dataset and an objective benchmark are vital preconditions for solid research and evaluation towards this goal. We show that existing datasets exhibit various problems, such as the quality of the commit selection, small sample sizes, duplicates, privacy issues, and missing licenses for redistribution. This can lead to unusable models and skewed evaluations, where inferior models achieve higher evaluation scores due to biases in the data. We compile a new large-scale dataset, CommitBench, adopting best practices for dataset creation. We sample commits from diverse projects with licenses that permit redistribution and apply our filtering and dataset enhancements to improve the quality of generated commit messages. We use CommitBench to compare existing models and show that other approaches are outperformed by a Transformer model pretrained on source code. We hope to accelerate future research by publishing the source code.**

***Index Terms*—Commit Message Generation, Dataset, Natural Language Processing**

I. Introduction

Developers generally appreciate informative commit messages that verbally describe the changes made to a software code base when a new version (commit) is recorded in a version control system (VCS). While well-crafted commit messages can be of immense help to understand the history and design decisions of a software project, many developers find them tedious to write and thus they are often neglected in practice [1], [2]. Another challenge is that developers have different standards, qualitative expectations, and styles regarding the commit message.

A system that automatically suggests informative and consistent commit messages would have the potential to save valuable time and enhance the quality of over 100 million commit messages a day.[1]

In recent years, a number of existing artificial intelligence models, originally developed for natural language processing (NLP) tasks such as machine translation or summarization, have been trained to translate code into text [3]–[5] or vice versa [6]–[8], but also to translate and summarize version differences into commit messages [9]–[14]. Most research on commit message generation is based on datasets that have become outdated, either due to their small size, their low quality, or their lack of concern for privacy, reproducibility, and license restrictions. These limitations point towards the need for better datasets for the task of commit message generation.

We present `CommitBench`, a new dataset for training and evaluating AI models. This datasets not only adopts existing best practices regarding the repository selection and filters, but also introduces novel filtering techniques. These contributions aim to enhance the quality of the dataset, while considering factors such as privacy, reproducibility, and licenses. We embed CommitBench into the research landscape by training and evaluating models on a selection of existing commit message generation datasets as well as ours.

The dataset is available on Zenodo[2] and HuggingFace[3][4]. The source code for the dataset generation is available on GitHub.[5]

II. Problem Statement

Given two versions of the same source code, the first being the original source code and the second an adapted version of the first with t blocks of code changes, a *diff* is a short text that summarizes these changes. A single diff consists of t blocks of changed code lines along with the original ones as well as extra context, typically three lines before and after the change.

The task of commit message generation can be phrased as a sequence-to-sequence task: Given a complete textual diff consisting of a sequence of tokens $x_1, \ldots, x_n$ of length $n \in \mathbb{N}$, generate a sequence $y_1, \ldots, y_m$ of $m \in \mathbb{N}$ tokens describing the changes in natural language. The descriptions are expected to be informative, yet also short enough to allow developers to quickly skim long commit histories. The input length in the form of source code changes is usually substantially longer than the resulting commit message, i.e., $n \gg m$.

[1]https://octoverse.github.com/2019/ (visited on January 12, 2023) and https://evansdata.com/press/viewRelease.php?pressID=278 (visited on January 12, 2023)

[2]https://zenodo.org/records/10497442
[3]https://huggingface.co/datasets/maxscha/commitbench
[4]https://huggingface.co/datasets/maxscha/commitbench_long
[5]https://github.com/maxscha/commitbench

III. Related Work

A. Commit Messages

Commit messages play an important role in software engineering. One study investigated which factors contribute to a high quality commit message, and which are detrimental, according to developers [15]. The study finds that, while there is currently no universal standard for commit messages, most developers agree that not only should the commit message describe *what* changes are made, but also *why* they were done. The authors analyzed five open source software projects and their human-generated commit messages, concluding that on average 44% of all commit messages are in need of improvement. They published a machine learning model trained on this data to classify good commit messages.

The negligence of software developers, who often fail to take the time to write high-quality commit messages, along with advances in machine learning, has led to notable interest in devising systems to automatically suggest commit messages.

B. Commit Generation Approaches

Commit message generation as a machine learning task is highly related to the NLP tasks of machine translation and summarization, although other approaches also exist. First, the meaning of the changes needs to be inferred, then the inferred information is translated into natural language and summarized to retain just a concise description.

1) Text-Based Approaches: Many approaches treat commit message generation as a sequence-to-sequence task, for which NLP approaches based on deep neural networks can readily be adopted with just minimal changes. The diff is treated as an input text, and the commit message as the corresponding desired output text. One of the first to apply deep learning to commit message generation, Jiang et al. [16] used an Attentional RNN Encoder–Decoder, directly taken from a *Neural Machine Translation* (NMT) framework, to translate the input diff to a commit message. Their dataset $\mathit{CommitGen}_{\mathit{Data}}$ was created based on findings of their previous paper [17], which studied the differences between auto-generated messages and human-written messages. They found that 82% of human-written messages only contain one line of text and that the majority of these follow a “verb+object” format.

Analyzing the results from Jiang et al. [16], Liu et al. [9] found that the generated diffs on the test set share a high similarity to the ones seen during training. Additionally, they identified roughly 16% as bot or trivial messages and subsequently removed them from the original dataset to generate a cleaned version. They proposed a nearest neighbor approach, $\mathit{NNGen}_{\mathit{Model}}$, which outperforms the NMT-based approach [16] on CommitGen and on their NNGen dataset. They concluded that traditional retrieval approaches are also feasible for commit message generation, especially on a small dataset.

To improve the prediction of out-of-vocabulary words, Liu et al. [11] proposed *PtrGNCMsg*, an approach based on pointer-generation networks. They used the original CommitGen dataset [16], but also created an additional dataset on top of it, with another 1,000 Java projects not originally in CommitGen. Due to its small size, large overlap with CommitGen, and minimal use in later research, we do not discuss it further. Their results slightly outperform the CommitGen results.

Creating an LSTM-based encoder–decoder model, Pravilov et al. [18] made use of unsupervised pretraining tasks to learn meaningful embeddings for code edits. As one of their downstream tasks, they evaluated using the embeddings for commit message generation and were able to achieve a similar performance as the previous supervised models [11], [16].

After pretraining a Transformer model and learning contextualized code representations using a self-supervised code prediction task, Nie et al. [13] subsequently fine-tuned on the CommitGen dataset for commit message generation. Their results indicate that their suggested pretraining tasks are helpful for commit message generation, but they did not release any code to reproduce their results.

Elnaggar et al. [19] transfer previous research [20] into the area of NLP on source code. They used existing datasets made for different tasks, pre-training, transfer learning, and fine-tuning to develop their unified encoder–decoder Transformer model *CodeTrans*. Their model was shown to outperform previous approaches across all six trained tasks, among them commit message generation.

Jung et al. [21] presented *CommitBERT*, a CodeBERT [22] model fine-tuned for commit message generation. The model uses a pretrained CodeBERT encoder and a freshly initialized Transformer-based decoder. To evaluate his approach, the author created a dataset based on the CodeSearchNet [23] repository selection. The results showed that a pretrained Transformer-based model achieves higher scores compared to a non-pretrained one, but the model was not compared to other baseline models.

2) Structure-Based Approaches: A number of recent approaches explore the use of *Abstract Syntax Trees* (AST) instead of unstructured text input [12], [14], [24]. A source code's structure can be parsed into a tree-based code representation often used by compilers. Analogous to natural language, the syntactic rules that define an AST differ amongst programming languages.

Xu et al. [12] argued that code structure should explicitly be taken into consideration. They proposed *CoDiSUM*, a custom encoder–decoder model with copying mechanisms. They used the CommitGen dataset, showing that the performance benefits from additionally modeling code structure. However, incorporating programming language-specific features requires more work to adapt the model to other languages than text-only approaches.

In a recent approach incorporating ASTs, Dong et al. [24] focused on more fine-grained code change representations. They combine this with a Transformer decoder to generate the final commit messages. Although they provide an extensive evaluation, they train on the older and noisy CommitGen dataset. Additionally, their current implementation is limited to Java, due to their reliance on ASTs as inputs. Adapting to a new programming language would take significant effort, unlike with text-based approaches.

IV. Previous Datasets

The choice of dataset greatly affects what aspects a model learns and hence how it performs across different benchmarks, even within the same task. In Table I, we give an overview of previous datasets [9], [11], [12], [16], [21], [25], [26] as well as concurrent work [27], [28] with regard to a number of important attributes. We argue that all previous datasets on commit message generation disregard at least one – often many more – of the desiderata for a responsible, high-quality dataset. Out of these seven existing datasets, only three use any additional filters for quality control (CommitGen, NNGen, MCMD); only two are currently reproducible (MultiLang, MCMD); only one filters for permissively licensed repositories (CommitBERT). None have taken any privacy-preserving measures. In light of the ongoing academic [29], legal [30]–[32], public [33], and political [34], [35] debates on privacy and license issues, these factors cannot be ignored. Another important aspect is that the growing need for reproducibility and extensive meta-data for training and evaluation datasets [36], [37]. Providing rich context on the data collection and processing steps can facilitate fairer, reproducible and privacy-aware datasets and can guide dataset creators to be more intentional in their design.

Another limitation is that most datasets are Java-only. Even the rare exceptions of datasets that do include multiple programming languages [21], [25] generally do not verify the commit's programming language at the commit level, but merely at the level of entire repositories. It is therefore easy for commits in other programming languages to slip in, introducing noise. Finally, a limiting factor in previous datasets is the small repository selection. MCMD [26] draws its almost 2M commits from only 500 repositories, so that a large number of commits within training, validation, and test sets will not be independent of each other. The authors find that the models' performance degrades by up to 73.41% when they cleanly split projects between training and test set.

From the widespread lack of reproducibility and quality filtering all the way to license issues and small dataset sizes, it becomes obvious that all previous datasets for commit message generation fail to holistically satisfy all relevant desiderata for a modern evaluation dataset. This underscores the pressing need for a comprehensive, well-curated, and privacy-aware dataset that can serve as a benchmark for future research in commit message generation. The ideal dataset should not only be diverse in terms of programming languages but also be meticulously verified at the commit level to ensure the highest quality. Moreover, the dataset should be expansive enough to encompass a wide range of repositories, ensuring that the training and evaluation data are representative of real-world scenarios.

We proceed by reviewing the previous datasets used for commit message generation individually in further detail:

a) CommitGen$_{Data}$ [16]: This is a small Java-only dataset that is based on the top 1,000 Java repositories. It uses a V-DO filter for qualitative commit filtering, but has several downsides such as trivial and bot-generated messages [10], and duplicates. It also lacks reproducibility, and comes with fixed pretokenization, missing licenses, and a short sequence length for diffs and messages. Since this remains the most used dataset for commit message generation, we include it in our evaluation.

b) NNgen$_{Data}$ [9]: A cleaned version[6] of CommitGen$_{\text{Data}}$. It removes trivial and bot commit messages, but otherwise suffers from the same limitations.

c) PtrGNCMsg$_{Data}$ [11]: A dataset[7] that uses the same preprocessing as CommitGen$_{\text{Data}}$, but includes the top 2,000 Java repositories, instead of the top 1,000. Since it is thus very similar to CommitGen$_{\text{Data}}$ in its shortcomings, we disregard it in our evaluation.

d) MutliLang [25]: A small dataset[8] created from twelve hand-selected repositories. It includes multiple programming languages. Due to its extremely small size, and because it is not publicly available, we do not regard this dataset as a good basis for evaluation and hence exclude it in our own experiments.

e) CoDiSum$_{Data}$ [12]: This is a Java-only dataset[9], which also uses the top 1,000 Java repositories as its data source. It is the only dataset that includes a file-level programming language filter, but lacks any other quality filtering, resulting in a larger dataset than datasets from similar data sources. Since its preprocessing is not customizable and the raw data is not provided, we disregard it in the evaluation.

f) CommitBert [21]: The repository selection of CommitBert[10] is based on that of *CodeSearchNet* [23], which includes six programming languages. It is pretokenized and only contains the actual code changes, not the context around them.

g) MCMD [26]: A large dataset[11] with five programming languages using the top 500 repositories on GitHub. Unlike other datasets, there is no sequence length filter, which makes the dataset unnecessary large. The pretokenization results in an average diff sequence length of 3,448. Processing sequences of that length constitutes a problem for most models. RNNs struggle with vanishing gradients, while Transformer-based models are limited by the quadratic runtime of their attention mechanism, with most models unable to surpass 1,024 tokens (e.g., BERT can process a maximum of 512 Tokens).

While this research was underway, several studies also introduced new datasets for commit message generation, which we list here for completeness but do not further analyze or incorporate into the remainder of this paper:

[6]https://github.com/Tbabm/nngen.git
[7]https://zenodo.org/record/2593787/files/ptrgn-commit-msg.zip
[8]https://github.com/epochx/commitgen
[9]https://github.com/SoftWiser-group/CoDiSum/blob/master/data4CopynetV3.zip
[10]https://github.com/graykode/commit-autosuggestions/tree/master/experiment/dataset
[11]https://zenodo.org/record/5025758

TABLE I
COMPARISON OF EXISTING DATASETS FOR COMMIT MESSAGE GENERATION WITH REGARD TO DATASET BEST PRACTICES, QUALITY, AND PROGRAMMING LANGUAGES. THE LAST COLUMN SHOWS OUR NEWLY DEVELOPED DATASET COMMITBENCH.

Criteria/Dataset	**Commit Gen**	**NNGen**	**PtrGNC Msg**	**Multi Lang**	**CoDi Sum**	**Commit Bert**	**MCMD**	**Commit Chronicle**	**Commit Pack**	**CommitBench (ours)**
Train	26,208	22,112	23,622	122,756	75,000	276,392	1,800,000	7,659,458	57,700,105	1,165,213
Valid	3,000	3,000	5,050	15,344	8,000	34,717	225,000	1,554,042	✗	249,689
Test	3,000	2,521	3,988	15,344	7,661	34,654	225,000	1,486,267	✗	249,688
Repositories	1,000	1,000	2,000	12	1,000	52k	500	11.9k	2.8m	72k
Reproducible	✗	✗	✗	✓	✗	✗	✓	✓	✓	✓
Deduplicated	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓
License-Aware Data Collection	✗	✗	✗	✗	✗	✓	✗	✓	✓	✓
Published License	✗	MIT	✗	✗	✗	Apache 2.0	✗	Various	MIT	CC BY-NC
Additional Filtering	✓	✓	✗	✗	✗	✗	✓	✓	✓	✓
Privacy Measures	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓
Programming Languages	Java	Java	Java	Java, C++ Python	Java	Java, Ruby JavaScript, Go, PHP, Python	Java, C#, C++, Python, JavaScript	20 Languages	350 Languages	Java, Ruby JavaScript, Go, PHP, Python

h) CommitChronicle [27]: A dataset[12] with various filters, incorporating 10.7 million commits from almost 12,000 repositories across 20 programming languages. Uniquely, the authors focus on preserving commit history, arguing that it helps to generate commit messages.

i) CommitPack [28]: Two datasets that focus on leveraging commits for instruction fine-tuning of LLMs. Those datasets consist only of commits that change a single file. The datasets focus on instruction tuning, but could be used for commit message generation as well.

Merge Commits, *Reverse Commits*, and *Binary Changes* are removed in all datasets.

V. COMMITBENCH

We present CommitBench, a new benchmark for commit message generation. Our focus is on providing researchers with a large, high-quality dataset that is compliant with ethical data standards such as considering licenses and privacy-sensitive content.

A. Data Acquisition

To compile a collection of commit messages, the first key step is to select source code repositories with relevant commits. For this, we draw on GitHub, by far the largest host of source code repositories.

With over 85 million new repositories on GitHub in 2022 alone[13], it is not practical to incorporate every repository. Furthermore, there should be a first qualitative repository selection. This not only refines the dataset, but also ensures that the commit messages are representative of best practices in the industry. A consistently maintained repository from a large organization, which is potentially used by many others, likely has better commit messages than a small repository uploaded by an individual for mere exploratory use.

Our repository selection is based on that of `CodeSearchNet` [23]. They collected all publicly available non-fork GitHub repositories that are used by at least one other project. Subsequently, all repositories without an MIT license, which explicitly permits the re-distribution of parts of the repository as long as the original copy of the license is distributed, are removed.

This is important so that CommitBench can be publicly distributed. Due to the scale of the dataset, it would be unfeasible to ask every author for their explicit consent to consider their commits.

B. Data Processing

Within each downloaded repository, for each commit, we extract the commit message, diff, commit hash, project name, organization, commit author, and committer. The diff is obtained in textual form using the default `git diff` command, ensuring consistency in the extraction process. The commit hash is stored for reproducibility, so that together with the repository organization and name, the commit can be traced back to the original repository. Only commits with one parent are extracted, since having more than one indicates a merge commit, which does not add any source code of its own. To reduce the complexity of further data processing, we only store commits with a diff smaller than 1MB, as larger diffs typically do not change source code but data. Additionally, commits with excessively large diffs can skew the analysis and may not provide meaningful insights into typical coding practices or patterns.

C. Filtering Methods

Even among high-quality repositories, not every commit is well-suited as training or evaluation data. Table II shows the amount of commits affected by the different filters. In the following, we describe filtering techniques invoked to ensure a high-quality dataset. It is worth noting that there is an inherent overlap between some filters. For instance, the sequence length filter for diffs and the source code language filter share an overlap of 7,809,633 commits. This is because many changes unrelated to programming code, such as configurations or documentation, can be quite verbose.

[12]https://zenodo.org/records/8189044
[13]https://octoverse.github.com/ (visited on June 1, 2023)

TABLE II
COMMITS AFFECTED BY SPECIFIC FILTERS FROM THE UNFILTERED DATASET. NOTE THAT A SINGLE COMMIT CAN BE COUNTED BY MULTIPLE FILTERS SIMULTANEOUSLY.

Filter	Number of Commits
No filter	23,284,371
Bot commits	807,335
Binary commits	4,026,669
Commit messages < 8	8,070,122
Commit messages > 128	783,232
Diffs > 512	13,296,820
Trivial messages	270,330
Revert commits	65,868
Commit message language	193,695
Source code language	11,252,158
Commits left after all filters	1,664,590

a) Binary and File Mode Commits: Commit sub-changes can be categorized into three types: binary changes, file mode changes, and textual changes. A diff for a binary commit only indicates what file has been changed, but not what content changed. Due to the missing semantic information in the diff, it is impossible to discern enough meaning for a commit message from the commit diff, even for a professional developer, making commits for binary diffs unsuitable for conventional commit message generation.

File mode changes, such as changes in a file's permission, represent a different challenge. While these changes are technically straightforward, they often lack the contextual and semantic information required for meaningful commit message generation. For instance, a change in file permissions from "read-only" to "read-write" can easily be described in textual form using simple patterns, but the underlying reason might not be evident from the diff alone.

Hence, our work focuses on textual changes. Binary as well as file mode changes were excluded from the dataset.

b) Sequence Length: As with most sequence-to-sequence tasks, the length of individual sequences is unevenly distributed, usually skewed towards shorter sequences. Current model architectures are limited with regard to the sequence length. They either have a predefined maximum number of tokens they can process, e.g., attention-based models like BERT [38] with 512 tokens, or their performance degrades with long inputs, e.g., for RNN-based models due to vanishing gradients. Our experiments showed that a code-based tokenizer, such as the one of CodeTrans [19], would reduce the sequence length for the diffs by 15%, compared to the T5 tokenizer. Nevertheless, for a general benchmark focusing on existing models, we used the T5 tokenizer to determine the sequence length and removed diffs and messages longer than 512 tokens. Additionally, commit messages shorter than eight tokens were removed, since such messages are usually of poor quality, lacking sufficient information on what was changed and why. We provide detailed statistics about the sequence length in Table III.

To evaluate long-sequence models, we additionally release a dataset version with a longer maximum sequence length of 2,048 tokens.

TABLE III
STATISTICAL INFORMATION ON THE COLLECTED COMMITS BEFORE FILTERING FOR MESSAGE AND CODE DIFF LENGTH USING THE T5 TOKENIZER.

Statistic	Message Length	Code Diff Length
Mean	27.52	4 129.75
Median	11	662
Standard Deviation	243.10	15 904.66
Variance	59 098.21	252 958 201
Minimum Value	0	0
Maximum Value	619 941	499 044
Range	619 941	499 044
25th Percentile	6	286
50th Percentile	11	662
75th Percentile	20	2046

c) Trivial Messages: Trivial messages are commit messages that fail to convey significant meaning regarding what exactly was changed in the source code and why the change occurred. Training on these examples encourages non-informative commit messages. For CommitBench, those were derived by observing common patterns in non-informative commit messages over a range of projects. In the following, we list the regular expressions invoked to recognize trivial messages.

- `update changelog v?[\d*\.]*`
- `prepare version v?[\d*\.]*`
- `bump version v?[\d*\.]*`
- `modify makefile`
- `update submodule .*`

d) Bot Commits: Previous research [10] found that a non-negligible number of commits in the `CommitGen` dataset was created by development tools, which they called bot commits. Commits from bots were identified through the commit author's name, author's e-mail, committer's name, or committer's e-mail address. If any of these contained the word "bot", the commit was removed from the dataset. This filtering method, while straightforward, may not capture all bot-generated commits, but it constitutes a reasonable attempt at reducing their prevalence in the dataset.

e) Revert Commits: A revert commit will undo a previous commit by creating a new commit that reverts all the previous commit changes. Since Git automatically generates revert messages, they were excluded from the dataset. Revert commits can easily be identified by the usage of *Revert Commit HASH* in the commit message.

f) Commit Message Language: Despite the focus on English projects, there are still commits with non-English messages. We detected these with `fasttext` [39], [40] language identification and removed them if its confidence for English is lower than 50%. This ensures a more consistent dataset, minimizing potential biases or inaccuracies arising from non-English commit messages.

g) Source Code Language: The original `CodeSearchNet` dataset filters out repositories that are not predominantly using one of the six considered programming languages. Despite this, many commits from this repository selection include changes in files that do not contain any source code written in those programming languages. Instead, many commits only change configuration or documentation files. As Table II shows, this includes almost half the commits in the unfiltered data. We therefore removed commits where less than 50% of the associated file name extensions were among the following: `php`, `rb`, `go`, `js`, `py`, or `java`. This filtering criterion was established to maintain the dataset's focus on the primary programming languages and to ensure the relevance of the commits to code-centric tasks.

h) Duplicate Detection: Duplicates, especially between the training and validation/test split, can have adverse effects on the evaluation of models [41]. CommitBench is deduplicated by removing samples with the same diff. Furthermore, the presence of duplicates can lead to overly optimistic performance metrics, as models might simply memorize patterns rather than genuinely learning to generalize. Hence, we hypothesize that the rigorous deduplication process in CommitBench is crucial for obtaining a realistic assessment of model capabilities.

i) Irrelevant Information: Many commit messages include information that refers to a different platform, such as issue IDs and URLs. We replaced such information with unique tokens. Version numbers were also replaced in the commit message, since they can easily be extracted from the diff. This standardization ensures a more generalized representation, allowing models to focus on the core content of the commit message without being distracted by platform-specific details.

j) Privacy: Extra effort was put into removing privacy-sensitive data, such as full names and e-mails. Full names were identified via named entity recognition using *FLAIR* [42], while e-mail addresses were identified via pattern matching. Both were replaced with special tags. We found that often the identified names and e-mails follow phrases such as *Thanks to* or *Contributed by*. We therefore removed all such lines in a commit message. Additionally, we removed all author and committer names in the released dataset.

D. Final Dataset

After filtering, the dataset is randomly partitioned into a 70%/15%/15% split. We evaluate an alternative repository-based split strategy in Section VII-C. Finally, the data is distributed as a `CSV` file, containing the project name, hash, commit message, diff, and split. This format has the advantage of retaining all pertinent data in a single file, facilitating data processing and sharing. For each commit and its commit message, we provide the respective programming language via file name extension analysis.

Our final dataset consists of 1,664,590 high-quality commits from 38,578 organizations and 71,676 unique projects, changing 1,781,414 files. To compare, applying our extensive filtering to MCMD, another large dataset, results in fewer than 400k total data samples, instead of its original 2.25M. Our commits stem from 169,316 unique authors and all repositories were downloaded on September 1, 2022. Table IV presents the programming language variety in the final dataset.

TABLE IV
OVERVIEW OF SPLIT BY PROGRAMMING LANGUAGE IN THE DIFF FOR COMMITBENCH. A SINGLE COMMIT THAT CONTAINS CHANGES IN FILES WITH MULTIPLE PROGRAMMING LANGUAGES WILL BE COUNTED TOWARDS EACH OF THEM.

Language	# Samples
Java	153,119
Ruby	233,710
Go	137,998
JavaScript	373,598
Python	472,469
PHP	294,394

We have maintained transparency in our process by storing the used repositories in a list. To trace back to original commit data, one can use the hash, name, and repository. Given that repositories can be deleted or modified, we also hold a backup copy of all the repositories used, available on request for privacy reasons. These measures ensure CommitBench remains a dependable tool for research in the future.

VI. EXPERIMENTAL SETUP

The comparability of previous work has suffered from inconsistent metrics (e.g., BLEU coming in many variations). We remedy this by training and evaluating all models on a selection of previous commit message generation datasets as well as ours, simultaneously embedding CommitBench into the research landscape. In the following we describe the experimental setup used to answer our research questions:

RQ1 How do various approaches, such as RNN-based, retrieval-based, and Transformer-based models, compare in terms of accuracy, diversity, and quality of generated commit messages?

RQ2 How does the split of a diverse dataset affect the accuracy of models?

RQ3 What are the effects of multi-language training compared to single-language training?

A. Model Setup

In our experiments, we benchmarked the following popular models, aiming at covering a diverse set of approaches: (1) CommitGen$_{\text{Model}}$ – the first commit message generation model designed for this task, which is now a ubiquitous baseline. (2) NNGen$_{\text{Model}}$ – the nearest-neighbor approach that came shortly after. (3) T5, a prominent Transformer model. (4) CodeTrans, a recent model with the same architecture as T5, but pretrained on source code tasks. For each Transformer-based model, we evaluated two different model sizes. Table V gives a comparison of the number of parameters of the evaluated models.

Our aim was to train the models in a similar setup to their original implementations. We therefore made use of the existing code repositories[14]. We took the models with default settings

[14]NNGen: https://github.com/Tbabm/nngen
T5: https://huggingface.co/t5-small
CodeTrans: https://huggingface.co/SEBIS/code_trans_t5_small_transfer_learning_pretrain
CommitGen's code repository has since been deleted.

TABLE V
COMPARISON OF THE NUMBER OF MODEL PARAMETERS FOR THE EVALUATED MODELS.

Models	# Model Parameters
CommitGen	53 Million
NNGen	0
T5$_{\text{Small}}$	60 Million
T5$_{\text{Base}}$	220 Million
CodeTrans$_{\text{Small}}$	60 Million
CodeTrans$_{\text{Base}}$	220 Million

described in the respective papers. For T5- and CodeTrans-based models, the hyperparameters given in Table VI were used. All experiments are reported based on a single run due to the extensive training times of large model sizes.

TABLE VI
HYPERPARAMETERS USED FOR FINE-TUNING AND EVALUATING T5- AND CODETRANS-BASED MODELS.

Parameter	Value
Batch Size	512
Optimization Method	AdamW
Learning Rate	5×10^{-5} with linear decay
Gradient Clipping	1.0
Dropout Rate	0.1
Number of Epochs	40
Beam Size	5

While we provide an extensive overview of models used for commit generation, the list is non-exhaustive. This limitation is due to unavailable implementations or, in some cases, implementations that we estimated would need over two weeks to train.

B. Datasets

Apart from CommitBench, we included the original CommitGen dataset, due to its popularity in prior work. We further compared with NNGen$_{\text{Data}}$, since it is a cleaned version of CommitGen$_{\text{Data}}$. Lastly, we also incorporated MCMD, one of the largest commit message generation dataset.

C. Evaluation Metrics

Evaluating generated commit messages is a non-trivial task. Previous research [11], [16], [18], [19] has often used traditional machine translation metrics, such as BLEU [43], ROUGE-L [44], and METEOR [45], to compare the generated message to a reference message based on token overlap heuristics. This approach can lead to skewed results, as the set of valid commit messages for a single commit is often larger than in traditional translation tasks, yet only one reference message is provided.

Wang et al. [46] highlight the challenges associated with ensuring the semantic relevance of generated commit messages. Their work underscores the need for a more refined evaluation mechanism that can discern between semantically relevant and irrelevant commit messages. To overcome this challenge we use 4 different evaluation metrics, briefly described in the following:

BLEU [43] is a common evaluation metric in machine translation. It compares a candidate output with a set of gold standard references. BLEU scores range from 0 to 1, with higher scores indicating better overlap and 1 indicating a perfect match with the reference. Due to the fact that numerous variations of BLEU exist, reported scores can vary widely. We chose the implementation by Post [47], which is consistent with the WMT standard.

ROUGE [44] was originally developed to evaluate summarization methods. It measures the overlap between the n-grams in the generated text and the reference text. We specifically consider the ROUGE-L variant, which focuses on the longest common subsequence, as it captures fluent and coherent sequence generation. ROUGE scores also range from 0 to 1, with higher scores indicating better overlap with the reference. We use a popular implementation[15].

METEOR [48] is a machine translation metric originally designed to address problems with BLEU, such as its lack of synonym matching. It aims to obtain greater correlations with human evaluations. METEOR scores range from 0 to 1, with higher scores indicating better alignment with human judgments. We use the METEOR Universal variant [45].

C-GOOD [15] uses a Bi-LSTM model trained on human-annotated commit messages to evaluate if a commit message contains information reflecting the *why* and *what* underlying the change. Conveying this in commit messages is crucial for developers to grasp the context and purpose of a commit. C-GOOD reports the percentage of good commit messages in the generated predictions in the range from 0 to 1, with higher percentages indicating better quality. Notably, it is the only metric that does not use references for evaluation. C-GOOD evaluates a commit message in isolation, without considering the actual code changes made in the commit. This means that the evaluation does not take into account how well the commit message aligns with the specific code modifications, potentially overlooking the relevance and accuracy of the message in the context of the changes made. We use the original script and data provided by the authors.[16]

VII. RESULTS

We provide a high-level overview of the overall performance of current commit message generation in Table VII on the four commit message generation datasets. NNGen$_{\text{Model}}$ and the larger CodeTrans model obtain the strongest results across all datasets. The strong results of NNGen$_{\text{Model}}$ are remarkable, considering that it is a simple nearest-neighbor retrieval approach. An explanation could be that commits often follow similar use cases, enabling the model to retrieve many relevant neighbors. This phenomenon is likely most pronounced on the MCMD data, which is based on only 500 repositories, so that most commits in the test set match others from the same project observed during training. It is much less of an issue for NNGen$_{\text{Data}}$ and CommitBench, where for both NNGen$_{\text{Model}}$

[15]https://pypi.org/project/rouge-score/

[16]https://github.com/WhatMakesAGoodCM/What-Makes-a-Good-Commit-Message

TABLE VII
OVERVIEW OF TEST RESULTS COMPARING MODELS EACH TRAINED AND TESTED ON THE SAME DATASET, FOR FOUR DIFFERENT DATASETS. BEST EVALUATION RESULTS FOR EACH METRIC AND DATASET ARE IN **BOLD**.

Dataset	Metric	CommitGen	NNGen	T5$_{\text{Small}}$	T5$_{\text{Base}}$	CodeTrans$_{\text{Small}}$	CodeTrans$_{\text{Base}}$
CommitGen	METEOR	0.418	**0.428**	0.357	0.397	0.385	0.404
	ROUGE-L	0.373	0.382	0.354	0.409	0.405	**0.433**
	BLEU	0.375	**0.387**	0.223	0.283	0.295	0.314
	C-GOOD	**0.418**	0.417	0.381	0.344	0.354	0.361
NNGen	METEOR	0.309	0.325	0.291	0.326	0.315	**0.339**
	ROUGE-L	0.246	0.270	0.243	0.300	0.297	**0.330**
	BLEU	0.145	0.168	0.095	0.138	0.145	**0.172**
	C-GOOD	**0.439**	0.438	0.401	0.374	0.375	0.383
MCMD	METEOR	0.199	**0.289**	0.229	0.243	0.236	0.250
	ROUGE-L	0.130	**0.202**	0.139	0.166	0.163	0.181
	BLEU	0.102	**0.147**	0.072	0.095	0.098	0.114
	C-GOOD	0.489	0.538	**0.578**	0.555	0.556	0.570
CommitBench	METEOR	0.103	0.205	0.203	0.251	0.229	**0.259**
	ROUGE-L	0.094	0.146	0.148	0.204	0.187	**0.221**
	BLEU	0.023	**0.096**	0.021	0.047	0.033	0.053
	C-GOOD	0.478	0.639	**0.740**	0.629	0.649	0.632

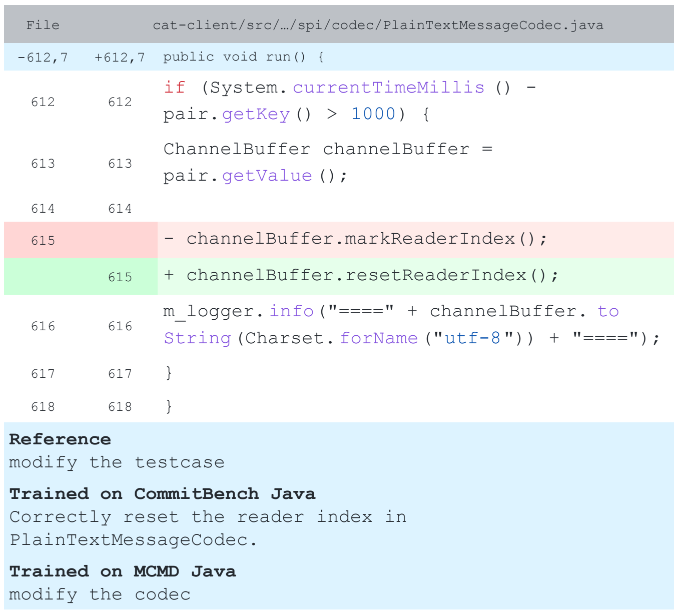
performs noticeably worse than CodeTrans. This suggests that these results are due to more varied commits from distinct project sources. The results on CommitGen$_{\text{Data}}$ are the highest, but this likely stems from the models being able to rely on surface patterns.

It is worth noting that CodeTrans models are generally strong performers across the board, but do not consistently achieve the highest C-GOOD scores. This indicates that solely large-scale code-specific pre-training is not sufficient for commit message generation. Additionally, it is crucial to recognize that the measurement of "good" commit messages, as quantified by the C-GOOD metric, is not infallible. Like all metrics, C-GOOD has limitations, such as disregarding the actual code change, and might not capture all nuances of what constitutes a high-quality commit message in every context. Therefore, while it provides a valuable indication of quality, it should be interpreted in conjunction with other metrics and qualitative assessments to obtain a comprehensive understanding of a model's performance.

In summary, NNGen$_{\text{Model}}$ and CodeTrans show the strongest performance in commit message generation across four datasets. NNGen$_{\text{Model}}$ does especially well on the limited MCMD dataset due to its nearest-neighbor retrieval approach. However, CodeTrans outperforms NNGen$_{\text{Model}}$ on the other datasets, indicating a better handling of diverse commits.

A. Output Diversity

Since we frame commit message generation as a language generation task, we also evaluate how the choice of dataset affects the diversity of model outputs. More diverse outputs imply more flexible models that genuinely attempt to interpret the input and produce a custom output sequence reflecting it. If the outputs are not diverse while performance is high, this can be a reflection of easy-to-abuse patterns in the training data. We use Self-BLEU [49], where a lower score means lower similarity with the other model outputs and thus a higher diversity.



```
File          cat-client/src/…/spi/codec/PlainTextMessageCodec.java
-612,7  +612,7  public void run() {
                if (System.currentTimeMillis () - pair.getKey () > 1000) {
                ChannelBuffer channelBuffer = pair.getValue ();

                - channelBuffer.markReaderIndex();
                + channelBuffer.resetReaderIndex();
                m_logger.info("====" + channelBuffer.toString(Charset.forName("utf-8")) + "====");
                }
                }
```

Reference
modify the testcase

Trained on CommitBench Java
Correctly reset the reader index in PlainTextMessageCodec.

Trained on MCMD Java
modify the codec

Fig. 1. Example of a diff, the reference commit message, and predicted commit messages from CodeTrans$_{\text{Base}}$ trained on the respective datasets. It can be observed that the better-generated commit message is further away from the reference message, which results in a lower evaluation score for this sample.

The results are given in Table VIII. We make two interesting

TABLE VIII
DIVERSITY OF MODEL OUTPUTS TRAINED ON ALL DATASETS, REPORTED WITH SELF-BLEU. LOWER MEANS MORE DIVERSE. BEST EVALUATION RESULTS IN **BOLD**.

Model	CommitGen	NNGen	MCMD	CommitBench
CommitGen	0.555	0.506	0.810	0.878
NNGen	0.490	0.412	0.345	0.348
$T5_{Small}$	0.614	0.539	0.652	0.423
$T5_{Base}$	0.579	0.510	0.591	0.388
CT_{Small}	0.576	0.488	0.594	0.418
CT_{Base}	0.539	0.468	0.569	0.382
Average	0.559	0.487	0.594	**0.473**

observations. (1) Training on CommitBench results in models having the most diverse outputs. A close second is $NNGen_{Data}$, tying into the observations made in the first experiment: that $NNGen_{Model}$ is more effective on $CommitGen_{Data}$ and MCMD, since it can rely on similar train and test sets. (2) $CommitGen_{Model}$'s outputs on MCMD and CommitBench exhibit a low degree of diversity. We believe that, due to $CommitGen_{Model}$'s limited complexity, the model is unable to capture the diversity of the larger CommitBench and MCMD datasets. The highly parametrized Transformer-based models can maintain diverse outputs on CommitBench, likely due to the large dataset size as well as the heterogeneous repository combination.

In short, sufficiently parametrized models trained on CommitBench have the most diverse output.

B. Qualitative Analysis

While manually reviewing the datasets, we noticed a significant discrepancy in the quality of commit messages. Specifically, in MCMD, many human-written commit messages do not adhere to the best practices of message composition. They often lack informativeness, being very short and sometimes vague. This is a serious problem, as models trained on MCMD tend to inherit these characteristics, often generating short and less informative commit messages.

In contrast, since we employ a filtering mechanism that removes commits with fewer than eight tokens in CommitBench, the models trained on this dataset tend to produce longer and more detailed messages. This distinction becomes evident when we compare the outputs. We provide an illustrative example in Figure 1. While the model trained on CommitBench gives a more informative and detailed output, the MCMD model's generation, albeit shorter, is closer to the human reference. This paradoxically would result in a higher evaluation score for the MCMD model. Such observations underscore the importance of dataset quality and curation. We take this as further validation of the need for extensive filtering and careful selection during dataset creation.

C. Repository-based Split

The choice of evaluation split can significantly influence the performance of a model. One common approach to splitting the data is based on the repository from which a sample originates, instead of randomly splitting the samples on the commit-level. For instance, in the case of MCMD [26], the authors report that a change of splitting strategy from random to repository-based in their evaluation split led to a drastic 53% drop in performance.

As illustrated in Table IX, Transformer-based models were trained on CommitBench using two distinct evaluation splits. One version employed a random split, while the other adopted a repository-based division.

We observe that our dataset, CommitBench, is sufficiently diverse to mitigate the impact of splitting strategy. In comparison to MCMD, the difference between a random split and a repository-based split in CommitBench is negligible. This is underscored by the fact that, despite having a comparable number of samples, CommitBench sources its data from nearly 150 times more repositories than MCMD, with 72,000 repositories as opposed to MCMD's 500. Consequently, the potential for code overlap between the training and testing sets is substantially reduced in CommitBench, making it a more robust dataset.

The results in Table IX reflect this and thus provide compelling insights into the robustness of CommitBench. While there is a noticeable performance drop in models evaluated using a repository-based split compared to a random split, the decline is not as pronounced as the one reported for MCMD. The slight performance drop in the repository-based split can be attributed to the inherent challenges of generating commit messages for previously unseen repositories. The fairly modest decrease in performance when the data is split based on repositories underscores the inherent diversity of CommitBench.

In short, a more sophisticated splitting strategy for commit message generation is optional when the sourced data is sufficiently diverse.

D. Multi-Language Impact

Similar to natural languages, there exist a variety of programming languages. While some are more low-level, i.e., more hardware-related, others are higher-level languages with more abstract concepts. We hypothesize that, analogous to natural language models, the per-language performance of dedicated monolingual models in high-resource settings may be better than that of multi-language models. To verify this, we trained all T5-based models on both: each programming-language-specific subset of CommitBench as well as jointly on all samples.

The results are reported in Table X. The multilingual setup consistently outperforms the monolingual one. We see several possible reasons. Learning from a diverse set of languages teaches the models fundamental principles all languages have in common. Drawing from multiple programming languages appears to equip the models with a broader syntactic and semantic understanding, enabling them to handle diverse coding patterns more effectively. Another aspect is the larger number of training updates when combining all training examples. We find the latter to more likely be the key reason, supported by

TABLE IX

RESULTS OF TRAINED AND EVALUATED T5-BASED MODELS ON THE COMMITBENCH DATASET. "RANDOM" IS THE STANDARD SPLIT, WHILE "REPO" SPLITS THE DATASET BASED ON REPOSITORIES.

	METEOR		ROUGE-L		BLEU		C-GOOD	
Split	**Random**	**Repo**	**Random**	**Repo**	**Random**	**Repo**	**Random**	**Repo**
T5_{Small}	.203	.186	.148	.123	.021	.014	.740	0.814
T5_{Base}	.251	.220	.204	.176	.047	.029	.629	0.669
$\text{CodeTrans}_{\text{Small}}$	.229	.0197	.187	.151	.033	.017	.649	0.740
$\text{CodeTrans}_{\text{Base}}$	.259	.217	.221	.186	.053	.028	.632	0.678

TABLE X

RESULTS WHEN TRAINING THE T5-BASED MODELS ON THE SIX INDIVIDUAL PROGRAMMING LANGUAGES IN COMMITBENCH. EVALUATION IS THEN DONE ON THE LANGUAGE-SPECIFIC SUBSET OF THE TEST SET AND COMPARED TO THE PERFORMANCE OF THE MODEL TRAINED ON ALL LANGUAGES ON THE SAME SUBSETS. THE RESULTS ARE REPORTED AS ROUGE-L SCORES. THE BETTER SCORE WITHIN EACH MONO- AND MULTI-LANGUAGE COMPARISON IS COLORED BLUE. BEST OVERALL RESULTS FOR EACH LANGUAGE IN **BOLD**.

Language	Java		Python		Go		JavaScript		PHP		Ruby	
Model	**One**	**All**	**One**	**All**	**One**	**All**	**One**	**All**	**One**	**All**	**One**	**All**
T5_{Small}	.130	.140	.151	.157	.131	.150	.116	.126	.124	.139	.170	.186
T5_{Base}	.177	.194	.219	.219	.179	.210	.175	.185	.180	.192	.220	.234
$\text{CodeTrans}_{\text{Small}}$	.167	.180	.189	.203	.157	.180	.147	.165	.163	.177	.209	.222
$\text{CodeTrans}_{\text{Base}}$	.203	**.214**	**.235**	**.235**	.209	**.226**	.194	**.201**	.209	**.213**	.241	**.249**

the fact that the differences in performance on Python are the smallest, which also has the most individual samples out of all languages (for an overview of samples per language, see Table IV).

Overall, multilingual models consistently outperformed monolingual ones, likely due to their broader syntactic and semantic understanding from diverse language exposure and more extensive training updates.

VIII. CONCLUSION

This paper presents CommitBench, a modern and comprehensive dataset for commit message generation. Reviewing existing datasets for the task, we identify critical weaknesses with regard to the quality of the data. Based on a discussion of best practices for dataset creation, we construct a number of preprocessing and filtering techniques to address these shortcomings and compile CommitBench. We unify previous research by comparing reproducible approaches for commit message generation, providing a consistent set of metrics across all datasets and models. Simultaneously, we embed CommitBench into the current research landscape by evaluating it against previous datasets, showing that training on CommitBench leads to models producing more diverse outputs and generating new baseline values for reference in future work. We show that fine-tuning Transformer-based models outperforms other approaches, and cross-programming-language training yields improved results.

A limitation of automatic commit message generation is the need for a metric to measure commit message quality. As we have shown, existing commit messages are occasionally not a good evaluation reference. In addition, while the correct answers for translation tasks are often straightforward, there can be fairly diverse yet valid commit messages. Furthermore, the context in which a commit is made, including the developer's intent and the broader project goals, can greatly influence the ideal message. This adds another layer of complexity to the evaluation, as understanding this context is often beyond the scope of automated systems. However, a direct human evaluation would require a large pool of programming experts on diverse software topics who would need to spend substantial time studying the broader context of each commit in a repository in order to adequately judge the quality of commit messages.

To overcome the previous limitations, future work could incorporate the evaluation into programmers' workflows with an integrated feedback loop. This could aid in evaluating datasets and further enhance trainable metrics like C-Good. Further, CommitBench could be extended to a much larger set of programming languages and repositories. Additionally, evaluating AST-based commit message generation approaches would be viable, though they require significant work since those approaches are usually programming-language-specific and would need to be adapted for every single evaluated language. Another promising research direction is using large language models, which we briefly tried but got inconsistent results, leaving space for further exploration. Additional contextual information, such as issues, could further help understand a developer's intent.

In conclusion, we hope that our public release of CommitBench at https://github.com/maxscha/commitbench will enable new research on these and other aspects of commit message generation.

References

[1] W. Maalej and H.-J. Happel, “Can development work describe itself?” in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, May 2010, pp. 191–200.

[2] 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)*, May 2013, pp. 422–431.

[3] D. Gros, H. Sezhiyan, P. Devanbu, and Z. Yu, “Code to Comment Translation: Data, Metrics, Baselining & Evaluation,” *arXiv:2010.01410 [cs]*, Oct. 2020.

[4] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, “Deep code comment generation,” in *Proceedings of the 26th Conference on Program Comprehension*. Gothenburg Sweden: ACM, May 2018, pp. 200–210.

[5] ——, “Deep code comment generation with hybrid lexical and syntactical information,” *Empirical Software Engineering*, vol. 25, no. 3, pp. 2179–2217, May 2020.

[6] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, “Mapping Language to Code in Programmatic Context,” *arXiv:1808.09588 [cs]*, Aug. 2018.

[7] L. Phan, H. Tran, D. Le, H. Nguyen, J. Anibal, A. Peltekian, and Y. Ye, “CoTexT: Multi-task Learning with Code-Text Transformer,” *arXiv:2105.08645 [cs]*, Jun. 2021.

[8] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, “CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation,” *arXiv:2102.04664 [cs]*, Mar. 2021.

[9] 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*. Montpellier France: ACM, Sep. 2018, pp. 373–384.

[10] K. Etemadi and M. Monperrus, “On the Relevance of Cross-project Learning with Nearest Neighbours for Commit Message Generation,” *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pp. 470–475, Jun. 2020.

[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)*. Montreal, QC, Canada: IEEE, May 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*. Macao, China: International Joint Conferences on Artificial Intelligence Organization, Aug. 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,” *arXiv:2007.06934 [cs]*, Feb. 2021.

[14] S. Liu, C. Gao, S. Chen, L. Y. Nie, and Y. Liu, “ATOM: Commit Message Generation Based on Abstract Syntax Tree and Hybrid Ranking,” *arXiv:1912.02972 [cs]*, Nov. 2020.

[15] 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*. Pittsburgh Pennsylvania: ACM, May 2022, pp. 2389–2401.

[16] S. Jiang, A. Armaly, and C. McMillan, “Automatically Generating Commit Messages from Diffs using Neural Machine Translation,” *arXiv:1708.09492 [cs]*, Aug. 2017.

[17] S. Jiang and C. McMillan, “Towards Automatic Generation of Short Summaries of Commits,” *arXiv:1703.09603 [cs]*, Mar. 2017.

[18] M. Pravilov, E. Bogomolov, Y. Golubev, and T. Bryksin, “Unsupervised learning of general-purpose embeddings for code changes,” in *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution*, ser. MaLTESQuE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 7–12.

[19] A. Elnaggar, W. Ding, L. Jones, T. Gibbs, T. Feher, C. Angerer, S. Severini, F. Matthes, and B. Rost, “CodeTrans: Towards Cracking the Language of Silicone's Code Through Self-Supervised Deep Learning and High Performance Computing,” *arXiv:2104.02443 [cs]*, Apr. 2021.

[20] 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,” *arXiv:1910.10683 [cs, stat]*, Jul. 2020.

[21] T.-H. Jung, “CommitBERT: Commit Message Generation Using Pre-Trained Programming Language Model,” *arXiv:2105.14242 [cs]*, May 2021.

[22] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “CodeBERT: A Pre-Trained Model for Programming and Natural Languages,” *arXiv:2002.08155 [cs]*, Sep. 2020.

[23] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “CodeSearchNet Challenge: Evaluating the State of Semantic Code Search,” *arXiv:1909.09436 [cs, stat]*, Jun. 2020.

[24] 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*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, May 2022, pp. 970–981.

[25] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, “A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes,” Apr. 2017.

[26] 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,” *arXiv:2107.05373 [cs]*, Jul. 2021.

[27] A. Eliseeva, Y. Sokolov, E. Bogomolov, Y. Golubev, D. Dig, and T. Bryksin, “From Commit Message Generation to History-Aware Commit Message Completion,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Luxembourg, Luxembourg: IEEE, Sep. 2023, pp. 723–735.

[28] N. Muennighoff, Q. Liu, A. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. von Werra, and S. Longpre, “OctoPack: Instruction Tuning Code Large Language Models,” Aug. 2023.

[29] F. Mireshghallah, F. Tramèr, H. Brown, K. Lee, and R. Shokri, “What does it mean for a language model to preserve privacy?” in *FaCCT*, 2022.

[30] Getty Images, “Getty Images statement,” https://newsroom.gettyimages.com/en/getty-images/getty-images-statement, Jan. 2023, accessed: 2023-05-22.

[31] M. Butterick, “Github copilot litigation,” https://githubcopilotlitigation.com/, Nov. 2022, accessed: 2023-05-22.

[32] J. A. Rothchild and D. Rothchild, “Copyright implications of the use of code repos- itories to train a machine learning model,” *Call for white papers on philosophical and legal questions around Copilot*, Feb. 2022. [Online]. Available: https://www.fsf.org/licensing/copilot/copyright-implications-of-the-use-of-code-repositories-to-train-a-machine-learning-model

[33] A. Hern and D. Milmo, ““I didn't give permission”: Do AI's backers care about data law breaches?” https://www.theguardian.com/technology/2023/apr/10/i-didnt-give-permission-do-ais-backers-care-about-data-law-breaches, Apr. 2023, accessed: 2023-05-22.

[34] S. Mukherjee, Y. C. Foo, and M. Coulter, “Eu proposes new copyright rules for generative ai,” https://www.reuters.com/technology/eu-lawmakers-committee-reaches-deal-artificial-intelligence-act-2023-04-27/, Apr. 2023, accessed: 2023-05-22.

[35] The Italian Data Protection Authority, “Artificial Intelligence: stop to ChatGPT by the Italian SA. Personal data is collected unlawfully, no age verification system is in place for children,” https://www.garanteprivacy.it/home/docweb/-/docweb-display/docweb/9870847, Mar. 2023, accessed: 2023-05-22.

[36] T. Gebru, J. Morgenstern, B. Vecchione, J. W. Vaughan, H. M. Wallach, H. D. III, and K. Crawford, “Datasheets for datasets,” *CoRR*, vol. abs/1803.09010, 2018. [Online]. Available: http://arxiv.org/abs/1803.09010

[37] E. M. Bender and B. Friedman, “Data statements for natural language processing: Toward mitigating system bias and enabling better science,” *Transactions of the Association for Computational Linguistics*, vol. 6, pp. 587–604, 2018. [Online]. Available: https://aclanthology.org/Q18-1041

[38] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” *arXiv:1810.04805 [cs]*, May 2019.

[39] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, “Bag of tricks for efficient text classification,” *arXiv preprint arXiv:1607.01759*, 2016.

[40] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, “FastText.zip: Compressing text classification models,” *arXiv preprint arXiv:1612.03651*, 2016.

[41] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Athens Greece: ACM, Oct. 2019, pp. 143–153.

[42] A. Akbik, T. Bergmann, D. Blythe, K. Rasul, S. Schweter, and R. Vollgraf, "FLAIR: An easy-to-use framework for state-of-the-art NLP," in *NAACL 2019, 2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, 2019, pp. 54–59.

[43] 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 on Association for Computational Linguistics - ACL '02*. Philadelphia, Pennsylvania: Association for Computational Linguistics, 2001, p. 311.

[44] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*, 2004, pp. 74–81.

[45] M. Denkowski and A. Lavie, "Meteor Universal: Language Specific Translation Evaluation for Any Target Language," in *Proceedings of the Ninth Workshop on Statistical Machine Translation*. Baltimore, Maryland, USA: Association for Computational Linguistics, 2014, pp. 376–380.

[46] 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)*, Mar. 2021, pp. 260–271.

[47] M. Post, "A Call for Clarity in Reporting BLEU Scores," in *Proceedings of the Third Conference on Machine Translation: Research Papers*. Belgium, Brussels: Association for Computational Linguistics, Oct. 2018, pp. 186–191.

[48] S. Banerjee and A. Lavie, "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments," in *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. Ann Arbor, Michigan: Association for Computational Linguistics, Jun. 2005, pp. 65–72.

[49] Y. Zhu, S. Lu, L. Zheng, J. Guo, W. Zhang, J. Wang, and Y. Yu, "Texygen: A benchmarking platform for text generation models," in *The 41st International ACM SIGIR Conference on Research; Development in Information Retrieval*, ser. SIGIR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1097–1100. [Online]. Available: https://doi.org/10.1145/3209978.3210080

APPENDIX A
DATA STATEMENT FOR COMMITBENCH

A. HEADER

- Dataset Title: CommitBench
- Dataset Curator: Maximilian Schall, Hasso Plattner Institute
- Dataset Version: 1.0, 01.01.2024
- Data Statement Author: Tamara Czinczoll, Hasso Plattner Institute
- Data Statement Version: 1.0, 16.01.2023
- Data URL: https://zenodo.org/records/10497442,https://huggingface.co/datasets/maxscha/commitbench
- Code URL: https://github.com/maxscha/commitbench

B. EXECUTIVE SUMMARY

We provide CommitBench as an open-source, reproducible and privacy- and license-aware benchmark for commit message generation. The dataset is gathered from github repositories with licenses that permit redistribution. We provide six programming languages, Java, Python, Go, JavaScript, PHP and Ruby. The commit messages in natural language are restricted to English, as it is the working language in many software development projects. The dataset has 1,664,590 examples that were generated by using extensive quality-focused filtering techniques (e.g. excluding bot commits). Additionally, we provide a version with longer sequences for benchmarking models with more extended sequence input.

C. CURATION RATIONALE

We created this dataset due to quality and legal issues with previous commit message generation datasets. Given a git diff displaying code changes between two file versions, the task is to predict the accompanying commit message describing these changes in natural language. We base our GitHub repository selection on that of a previous dataset, CodeSearchNet, but apply a large number of filtering techniques to improve the data quality and eliminate noise. Due to the original repository selection, we are also restricted to the aforementioned programming languages. It was important to us, however, to provide some number of programming languages to accommodate any changes in the task due to the degree of hardware-relatedness of a language. The dataset is provides as a large CSV file containing all samples. We provide the following fields: Diff, Commit Message, Hash, Project, Split.

D. DOCUMENTATION FOR SOURCE DATASETS

Repository selection based on CodeSearchNet, which can be found under https://github.com/github/CodeSearchNet.

E. LANGUAGE VARIETIES

Since GitHub hosts software projects from all over the world, there is no single uniform variety of English used across all commit messages. This means that phrasing can be regional or subject to influences from the programmer's native language. It also means that different spelling conventions may co-exist and that different terms may used for the same concept. Any model trained on this data should take these factors into account. For the number of samples for different programming languages, see Table IV.

TABLE XI
OVERVIEW OF SPLIT BY PROGRAMMING LANGUAGE FOR COMMITBENCH. A SINGLE COMMIT WHICH CONTAINS MULTIPLE PROGRAMMING LANGUAGE WILL BE COUNTED TOWARDS EACH OF THEM.

Language	# Samples
Java	153,119
Ruby	233,710
Go	137,998
JavaScript	373,598
Python	472,469
PHP	294,394

F. SPEAKER DEMOGRAPHIC

Due to the extremely diverse (geographically, but also socio-economically) backgrounds of the software development community, there is no single demographic the data comes from. Of course, this does not entail that there are no biases when it comes to the data origin. Globally, the average software developer tends to be male and has obtained higher education. Due to the anonymous nature of GitHub profiles, gender distribution information cannot be extracted.

G. ANNOTATOR DEMOGRAPHIC

Due to the automated generation of the dataset, no annotators were used.

H. SPEECH SITUATION AND CHARACTERISTICS

The public nature and often business-related creation of the data by the original GitHub users fosters a more neutral, information-focused and formal language. As it is not uncommon for developers to find the writing of commit messages tedious, there can also be commit messages representing the frustration or boredom of the commit author. While our filtering is supposed to catch these types of messages, there can be some instances still in the dataset.

I. PREPROCESSING AND DATA FORMATTING

See Section V for all preprocessing steps. We do not provide the un-processed raw data due to privacy concerns, but it can be obtained via CodeSearchNet (see Section A-D) or requested from the authors.

J. CAPTURE QUALITY

While our dataset is completely reproducible at the time of writing, there are external dependencies that could restrict this. If GitHub shuts down and someone with a software project in the dataset deletes their repository, there can be instances that are non-reproducible.

K. LIMITATIONS

While our filters are meant to ensure a high quality for each data sample in the dataset, we cannot ensure that only low-quality examples were removed. Similarly, we cannot guarantee that our extensive filtering methods catch all low-quality examples. Some might remain in the dataset. Another limitation of our dataset is the low number of programming languages (there are many more) as well as our focus on English commit messages. There might be some people that only write commit messages in their respective languages, e.g., because the organization they work at has established this or because they do not speak English (confidently enough). Perhaps some languages' syntax better aligns with that of programming languages. These effects cannot be investigated with CommitBench.

Although we anonymize the data as far as possible, the required information for reproducibility, including the organization, project name, and project hash, makes it possible to refer back to the original authoring user account, since this information is freely available in the original repository on GitHub.

L. METADATA

- License: Dataset under the CC BY-NC 4.0[17] license, code under the MIT license

M. DISCLOSURES AND ETHICAL REVIEW

While we put substantial effort into removing privacy-sensitive information, our solutions cannot find 100% of such cases. This means that researchers and anyone using the data need to incorporate their own safeguards to effectively reduce the amount of personal information that can be exposed.

N. OTHER

O. ABOUT THIS DOCUMENT

A data statement is a characterization of a dataset that provides context to allow developers and users to better understand how experimental results might generalize, how software might be appropriately deployed, and what biases might be reflected in systems built on the software.

This data statement was written based on the template for the Data Statements Version 2 schema. The template was prepared by Angelina McMillan-Major, Emily M. Bender, and Batya Friedman and can be found at https://techpolicylab.uw.edu/data-statements/ and was updated from the community Version 1 Markdown template by Leon Dercyznski.

APPENDIX B
DATA EXAMPLES

We show a selection of data examples of the CommitBench dataset in Figures 2, 3, and 4.

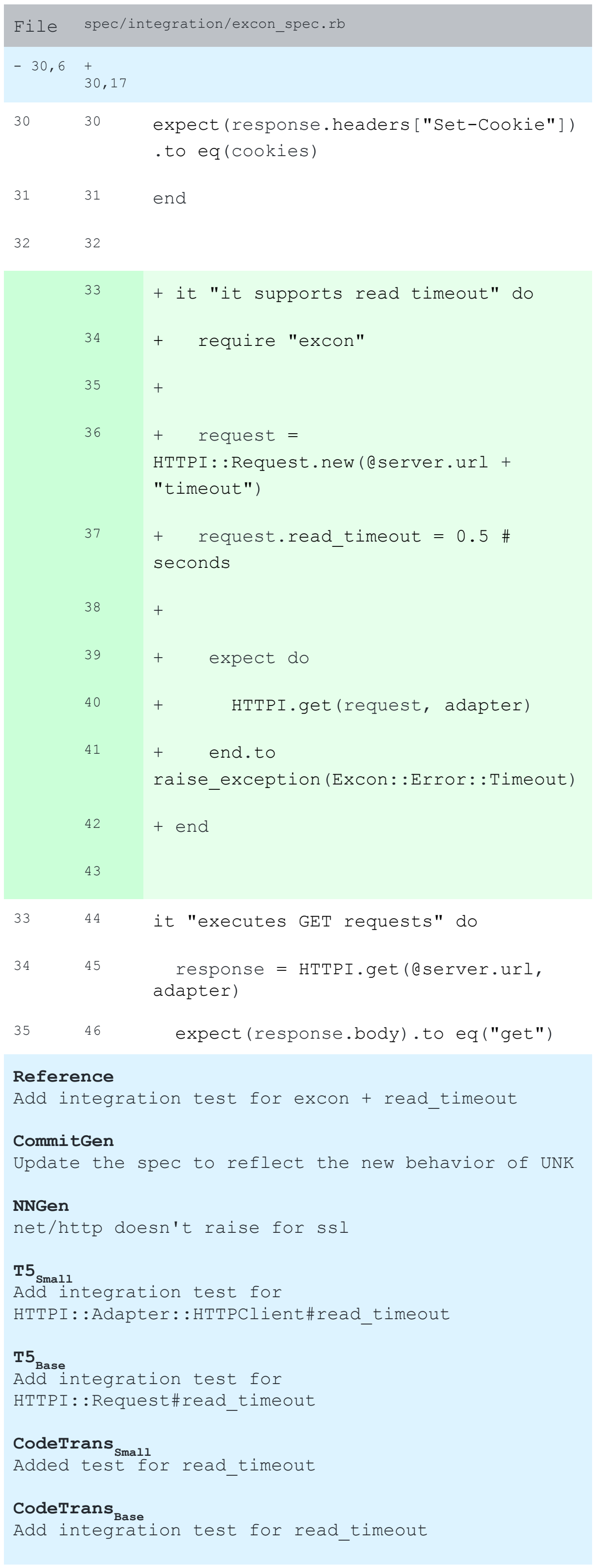
[17]https://creativecommons.org/licenses/by-nc/4.0/

APPENDIX C
COMPUTATIONAL BUDGET

As with most sequence-to-sequence tasks, the length of individual sequences is unevenly distributed, usually skewed towards shorter sequences. Current model architectures are limited with regard to the sequence length. Either they have a predefined maximum number of tokens they can process, e.g., attention-based models like BERT [38] with 512 tokens, or their performance degrades with long inputs, e.g., for RNN-based models due to vanishing gradients. To account for model and computational limitations we used the T5 tokenizer in our experiments to determine the sequence length, and removed diffs and messages longer than 512 tokens.

For CommitGen$_{\text{Model}}$ and NNGen$_{\text{Model}}$, we used the hyper-parameters reported by their original papers. CommitGen was trained on one NVIDIA A6000 with 50GB VRAM for 38h on CommitBench, while NNGen does not require a GPU and took 3h to compute the predictions for CommitBench.

These models were fine-tuned on four NVIDIA A100 with 40GB VRAM each. The *Small* models took 12h for fine-tuning on CommitBench, while the *Base* models took 35h for fine-tuning on CommitBench.



```
File    spec/integration/excon_spec.rb
- 30,6  +
        30,17
30      30      expect(response.headers["Set-Cookie"])
                .to eq(cookies)
31      31      end
32      32
        33      + it "it supports read timeout" do
        34      +   require "excon"
        35      +
        36      +   request =
                HTTPI::Request.new(@server.url +
                "timeout")
        37      +   request.read_timeout = 0.5 #
                seconds
        38      +
        39      +    expect do
        40      +      HTTPI.get(request, adapter)
        41      +    end.to
                raise_exception(Excon::Error::Timeout)
        42      + end
        43
33      44      it "executes GET requests" do
34      45        response = HTTPI.get(@server.url,
                adapter)
35      46        expect(response.body).to eq("get")
```

Reference
Add integration test for excon + read_timeout

CommitGen
Update the spec to reflect the new behavior of UNK

NNGen
net/http doesn't raise for ssl

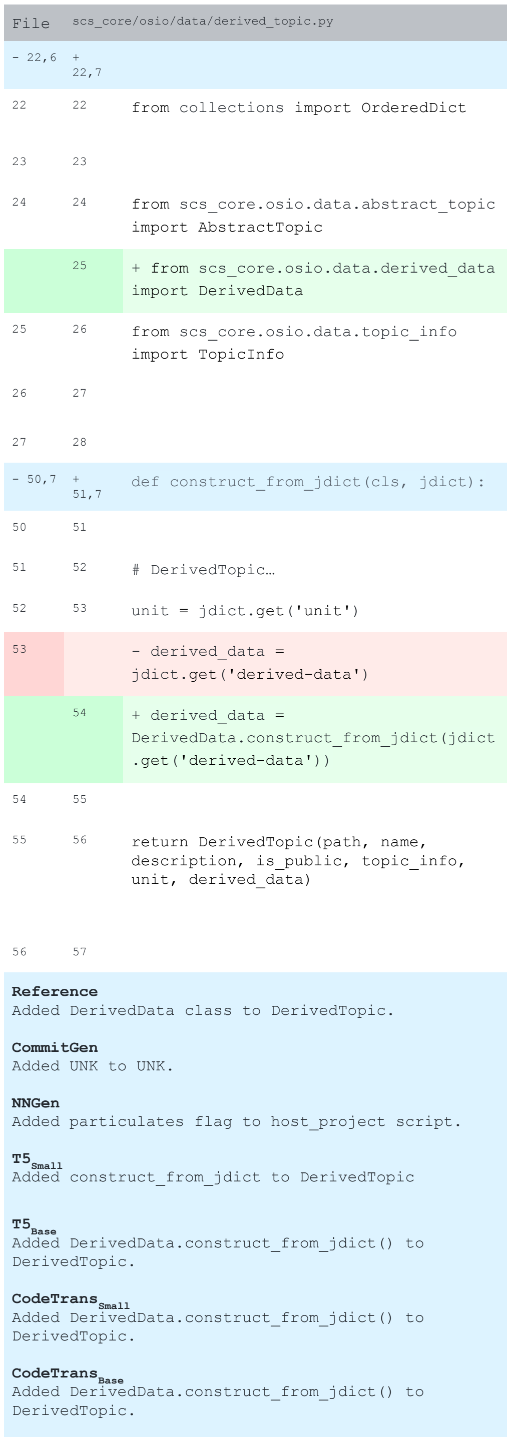
T5$_{\text{Small}}$
Add integration test for HTTPI::Adapter::HTTPClient#read_timeout

T5$_{\text{Base}}$
Add integration test for HTTPI::Request#read_timeout

CodeTrans$_{\text{Small}}$
Added test for read_timeout

CodeTrans$_{\text{Base}}$
Add integration test for read_timeout

Fig. 2. Sample of CommitBench-Test data, with predictions from CommitGen, NNGen, T5$_{\text{Small}}$, T5$_{\text{Base}}$, CodeTrans$_{\text{Small}}$, and CodeTrans$_{\text{Base}}$



```
File    scs_core/osio/data/derived_topic.py
- 22,6  +
        22,7
22      22      from collections import OrderedDict
23      23
24      24      from scs_core.osio.data.abstract_topic
                import AbstractTopic
        25      + from scs_core.osio.data.derived_data
                import DerivedData
25      26      from scs_core.osio.data.topic_info
                import TopicInfo
26      27
27      28
- 50,7  +       def construct_from_jdict(cls, jdict):
        51,7
50      51
51      52      # DerivedTopic…
52      53      unit = jdict.get('unit')
53              - derived_data =
                jdict.get('derived-data')
        54      + derived_data =
                DerivedData.construct_from_jdict(jdict
                .get('derived-data'))
54      55
55      56      return DerivedTopic(path, name,
                description, is_public, topic_info,
                unit, derived_data)
56      57
```

Reference
Added DerivedData class to DerivedTopic.

CommitGen
Added UNK to UNK.

NNGen
Added particulates flag to host_project script.

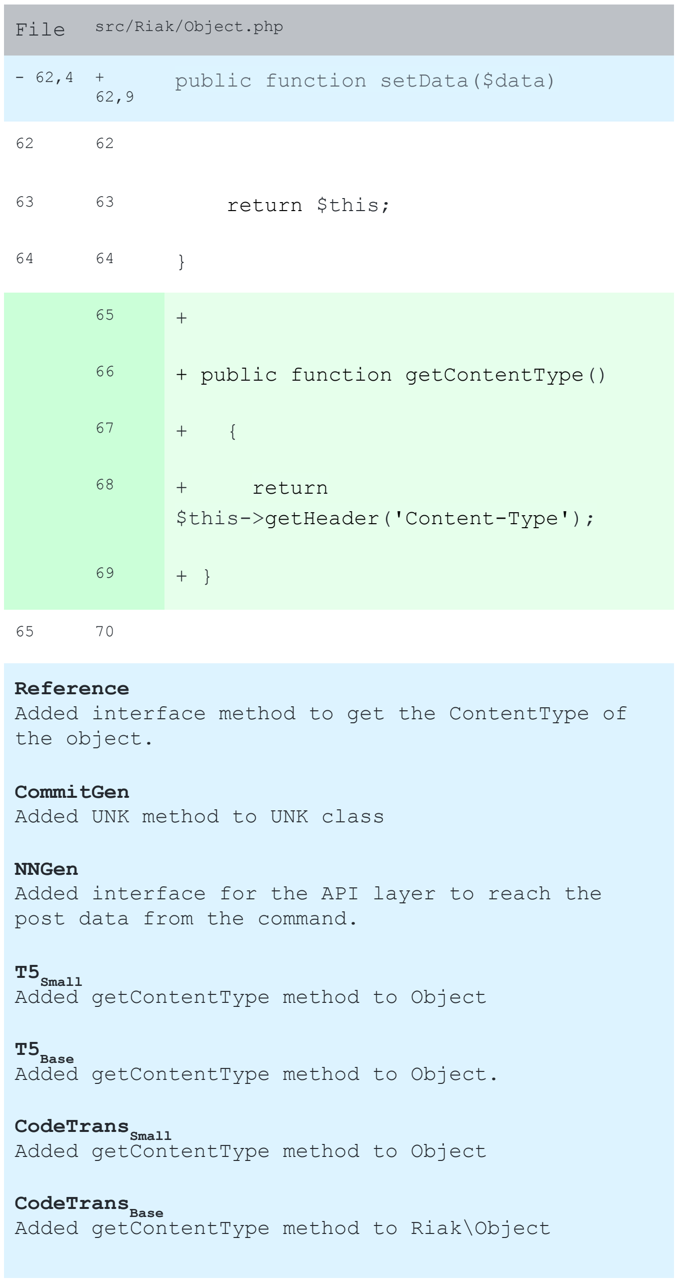
T5$_{\text{Small}}$
Added construct_from_jdict to DerivedTopic

T5$_{\text{Base}}$
Added DerivedData.construct_from_jdict() to DerivedTopic.

CodeTrans$_{\text{Small}}$
Added DerivedData.construct_from_jdict() to DerivedTopic.

CodeTrans$_{\text{Base}}$
Added DerivedData.construct_from_jdict() to DerivedTopic.

Fig. 3. Sample of CommitBench-Test data, with predictions from CommitGen, NNGen, T5$_{\text{Small}}$, T5$_{\text{Base}}$, CodeTrans$_{\text{Small}}$, and CodeTrans$_{\text{Base}}$



```
File      src/Riak/Object.php
- 62,4  + 62,9   public function setData($data)
62      62
63      63           return $this;
64      64       }
        65       +
        66       + public function getContentType()
        67       +    {
        68       +      return
                 $this->getHeader('Content-Type');
        69       + }
65      70
```

Reference
Added interface method to get the ContentType of the object.

CommitGen
Added UNK method to UNK class

NNGen
Added interface for the API layer to reach the post data from the command.

T5$_{Small}$
Added getContentType method to Object

T5$_{Base}$
Added getContentType method to Object.

CodeTrans$_{Small}$
Added getContentType method to Object

CodeTrans$_{Base}$
Added getContentType method to Riak\Object

Fig. 4. Sample of CommitBench-Test data, with predictions from CommitGen, NNGen, T5$_{Small}$, T5$_{Base}$, CodeTrans$_{Small}$, and CodeTrans$_{Base}$