CodeEditor: Learning to Edit Source Code with Pre-trained Models

JIA ALLEN LI, GE LI[*], ZHUO LI, and ZHI JIN[*], Key Lab of High Confidence Software Technology, MoE, School of Computer Science, Peking University, China

XING HU, Zhejiang University, China

KECHI ZHANG and ZHIYI FU, Key Lab of High Confidence Software Technology, MoE, School of Computer Science, Peking University, China

Developers often perform repetitive code editing activities (up to 70%) for various reasons (*e.g.,* code refactoring) during software development. Many deep learning (DL) models have been proposed to automate code editing by learning from the code editing history. Among DL-based models, pre-trained code editing models have achieved the state-of-the-art (SOTA) results. Pre-trained models are first pre-trained with pre-training tasks and fine-tuned with the code editing task. Existing pre-training tasks mainly are code infilling tasks (*e.g.,* masked language modeling), which are derived from the natural language processing field and are not designed for automatic code editing.

In this paper, we propose a novel pre-training task specialized in code editing and present an effective pre-trained code editing model named CodeEditor. Compared to previous code infilling tasks, our pre-training task further improves the performance and generalization ability of code editing models. Specifically, we collect lots of real-world code snippets as the ground truth and use a powerful generator to rewrite them into mutated versions. Then, we pre-train our CodeEditor to edit mutated versions into the corresponding ground truth, to learn edit patterns. We conduct experiments on four code editing datasets and evaluate the pre-trained CodeEditor in three settings (*i.e.,* fine-tuning, few-shot, and zero-shot). (1) In the fine-tuning setting, we train the pre-trained CodeEditor with four datasets and evaluate it on the test data. CodeEditor outperforms the SOTA baselines by 15%, 25.5%, and 9.4% and 26.6% on four datasets. (2) In the few-shot setting, we train the pre-trained CodeEditor with limited data and evaluate it on the test data. CodeEditor substantially performs better than all baselines, even outperforming baselines that are fine-tuned with all data. (3) In the zero-shot setting, we evaluate the pre-trained CodeEditor on the test data without training. CodeEditor correctly edits 1,113 programs while the SOTA baselines can not work. The results show that the superiority of our pre-training task and the pre-trained CodeEditor is more effective in automatic code editing.

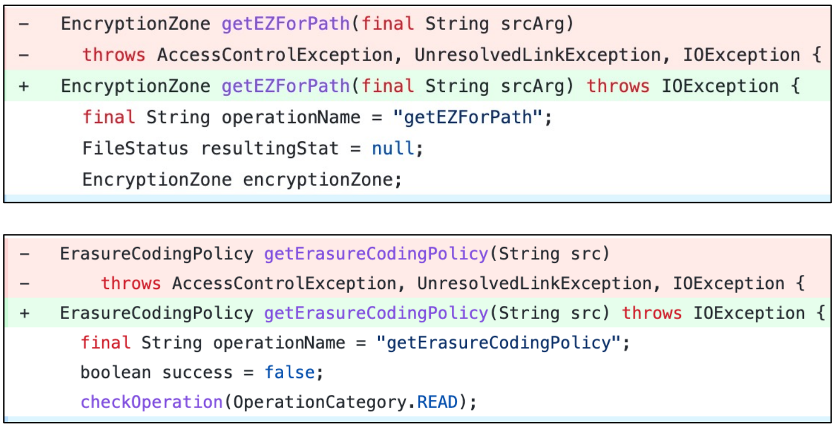
CCS Concepts: • **Computing methodologies** → *Neural networks*; *Natural language processing*; • **Software and its engineering** → **Automatic programming**.

Additional Key Words and Phrases: Source Code Editing, Pre-training, Deep Learning

[*]Corresponding author

Authors' addresses: Jia Allen Li, lijia@stu.pku.edu.cn; Ge Li, lige@pku.edu.cn; Zhuo Li, lizhmq@pku.edu.cn; Zhi Jin, zhijin@pku.edu.cn, Key Lab of High Confidence Software Technology, MoE, School of Computer Science, Peking University, No.5 Yiheyuan Road, Haidian District, Beijing, China; Xing Hu, Zhejiang University, No. 1689 Jiangnan Road, Gaoxin District, Ningbo, China, xinghu@zju.edu.cn; Kechi Zhang, zhangkechi@pku.edu.cn; Zhiyi Fu, fuzhiyi1129@gmail.com, Key Lab of High Confidence Software Technology, MoE, School of Computer Science, Peking University, No.5 Yiheyuan Road, Haidian District, Beijing, China.

0004-5411/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX



```
-   EncryptionZone getEZForPath(final String srcArg)
-     throws AccessControlException, UnresolvedLinkException, IOException {
+   EncryptionZone getEZForPath(final String srcArg) throws IOException {
      final String operationName = "getEZForPath";
      FileStatus resultingStat = null;
      EncryptionZone encryptionZone;
```

```
-   ErasureCodingPolicy getErasureCodingPolicy(String src)
-       throws AccessControlException, UnresolvedLinkException, IOException {
+   ErasureCodingPolicy getErasureCodingPolicy(String src) throws IOException {
      final String operationName = "getErasureCodingPolicy";
      boolean success = false;
      checkOperation(OperationCategory.READ);
```

Fig. 1. Two edits from a real-world software project [9] in GitHub. They both aim to remove redundant throw exceptions.

ACM Reference Format:
Jia Allen Li, Ge Li, Zhuo Li, Zhi Jin, Xing Hu, Kechi Zhang, and Zhiyi Fu. 2018. CodeEditor: Learning to Edit Source Code with Pre-trained Models. *J. ACM* 37, 4, Article 111 (August 2018), 22 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

To improve software systems' stability and maintainability, developers spend lots of effort (*e.g.*, more than six hours per week [2]) on editing their source code. For example, developers would modify identifier names or update an outdated API. A large-scale study in 2,841 Java projects [23] has shown that many edits (up to 70-100%) follow repetitive patterns. Figure 1 shows two edits from a real-world software project [9]. They both aim to remove redundant exceptions (*i.e.*, `AccessControlException` and `UnresolvedLinkException`) and share an edit pattern. However, manually designing these repetitive patterns can be tedious and error-prone [22, 28]. Thus, code editing models would be beneficial to save developers' effort by automating code changes learned from previous edit data.

Recently, deep learning (DL) techniques have been applied to automatic code editing. Among DL-based approaches, pre-trained code editing models [34, 37] have achieved state-of-the-art (SOTA) results on many benchmarks. Pre-trained models first are pre-trained with *self-supervised pre-training tasks*, and then fine-tuned with the *supervised code editing task*. Self-supervision means the labels of training samples are generated automatically without human annotations. Thus, a model can be pre-trained with a large amount of automatically generated data to learn linguistic and commonsense knowledge about the source code. Nowadays, existing code editing studies [34, 37] mainly use code infilling tasks (*e.g.*, mask language modeling) as the pre-training tasks. The code infilling tasks randomly mask some tokens or spans in a program and train a model to infill the masked content based on the contexts. Figure 2 (a) shows a code infilling example. The masked content (*i.e.*, `void`, `String[]`) is replaced with a specific token (*i.e.*, `MASK`) and highlighted. Although promising, code infilling tasks are derived from the natural language processing (NLP) field [6, 19] and are not designed for automatic code editing. Thus, there are still rooms to improve existing pre-trained code editing models.

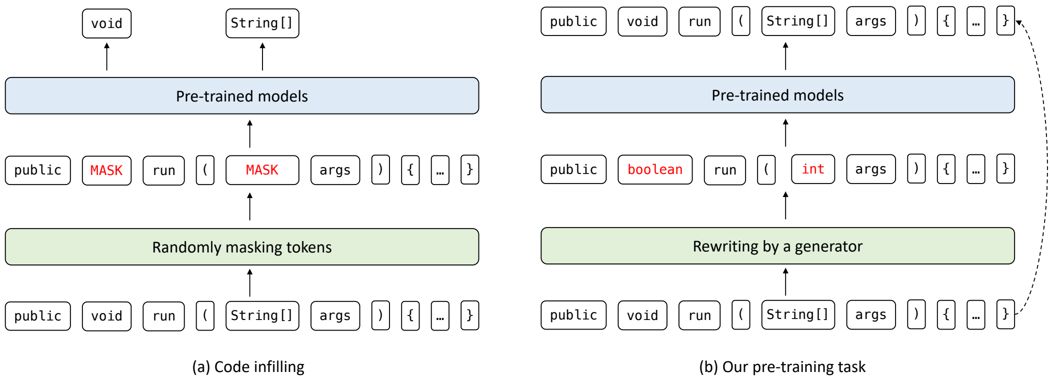


Fig. 2. The comparison of (a) code infilling tasks and (b) our pre-training task. Code infilling tasks randomly mask some tokens or spans in a program and train a model to infill the masked content based on the contexts. Our pre-training task uses a powerful generator to rewrite the code into a mutated version. Then, a model is trained to edit the mutated version into the original version.

In this paper, we propose to extend the conventional code infilling to a novel pre-training task specialized in code editing and present an effective pre-trained code editing model named CodeEditor. Specifically, we first collect lots of programs from open-source communities (*e.g.,* GitHub[1]). These programs have passed code reviews and can be viewed as the ground truth. We utilize a powerful *generator* to rewrite these programs into natural but inferior versions (aka mutated versions). Then, we pre-train the CodeEditor to edit the mutated code into the corresponding ground truth. Figure 2 (b) shows a training example in our pre-training task. The rewritten content is highlighted. In Figure 2 (b), the generator rewrites a program into a mutated version by modifying two tokens (`void` $\longrightarrow$ `boolean` and `String[]` $\longrightarrow$ `int`). Then, the pre-trained model is asked to edit the mutated code into the ground truth.

Compared to previous code infilling tasks, our pre-training task has two advantages: (1) **Our pre-training task improves the performance of code editing models.** The goal of code infilling tasks is to infill a given blank in the source code. Our pre-training task is more challenging and requires a high-level understanding ability to locate inferior parts and a strong generative ability to generate a better alternative. Thus, our pre-training task can provide strong supervision signals and further improves the performance of code editing models. (2) **Our pre-training task strengthens the generalization ability of code editing models.** Code infilling tasks are to predict some discrete tokens based on a masked program. Our pre-training task aims to transform a previous program into a new program by automating code changes and is closer to the real-world code editing task. Thus, our pre-training task endows the model with a practical code editing ability and strengthens the model's generalization ability in real-world code editing.

The key element in implementing our pre-training task is the generator for rewriting programs. Inspired by previous studies [5, 39], we utilize a powerful pre-trained language model for source code - CodeGPT [20] - as the generator. CodeGPT is trained on a code corpus consisting of 2.7 million files and is a SOTA language model for source code. Thus, CodeGPT can derive various informative code snippets that benefit our CodeEditor to learn meaningful and diverse edit patterns (*e.g.,* API updates, type/object changes, and identifier renaming). These edit patterns resemble the code changes in real-world code editing and thus strengthen the performance of pre-trained models in

[1]https://github.com/

code editing applications. Specifically, given an original program, we randomly select several spans and replace them with blanks. For each blank, CodeGPT can predict multiple plausible suggestions and ranks them based on probability. We consider the original span as the gold and use the first non-gold suggestion to complete the blank. After completing all blanks, we obtain a mutated version that needs to be edited. Its corresponding original code is the ground truth.

Following previous studies [4, 34], we conduct experiments on four public code editing datasets. We evaluate our CodeEditor in three settings (*i.e.,* fine-tuning, few-shot, and zero-shot settings) and employ three widely used metrics, including exact match (EM), CrystalBLEU [7], and edit distance. Based on experimental results, we summarize four findings. (1) In the fine-tuning setting, we train the pre-trained CodeEditor with four datasets and evaluate CodeEditor on the test data. In terms of EM, CodeEditor outperforms the SOTA baseline by 15%, 25.5%, 9.4%, and 26.6% on four datasets. (2) In the few-shot setting, we sub-sample (*e.g.,* 10%) the datasets, train CodeEditor with the limited data, and evaluate CodeEditor on the test data. CodeEditor still performs substantially better than all baselines, even outperforming baselines that are fine-tuned with all data. (3) In the zero-shot setting, we evaluate CodeEditor on the test data without training. Results show that CodeEditor successfully edits 1,123 programs, while baselines generate none of the correct code on four datasets. (4) We further conduct a case study by analyzing successful cases and failed cases of CodeEditor.

Our main contributions are outlined as follows:

- We propose a novel pre-training task for code editing that trains a model to edit an auto-rewritten mutated program into the ground truth. It can improve the performance and generalization ability of code editing models.
- We produce a large-scale dataset on our pre-training task and pre-train an effective pre-trained code editing model named CodeEditor.
- We fine-tune the pre-trained CodeEditor with four code editing datasets. Results show that CodeEditor is more effective by outperforming the SOTA baseline by up to 26.6%.
- We evaluate the pre-trained CodeEditor in zero-shot and few-shot settings. Results show that CodeEditor shows a strong generalization ability in both settings.

Data Availability. We open-source our replication package[2], including the datasets and the source code of CodeEditor, to facilitate other researchers and practitioners to repeat our work and verify their studies.

Paper Organization. Section 2 describes the background of our work. Section 3 presents our model CodeEditor. Section 4 and Section 5 describe the design and results of our study, respectively. Section 6 and Section 7 discuss some results and describe the related work, respectively. Section 8 concludes the paper and points out future directions.

2 BACKGROUND

2.1 Code Editing

Generating the source code using deep learning models has been explored in previous studies [31, 38]. These studies aim to build a model $p(\boldsymbol{y}|\boldsymbol{x})$ that generates the code $\boldsymbol{y}$ based on the given contextual information $\boldsymbol{x}$ (*e.g.,* natural language requirements). In this work, code editing is a special case of $p(\boldsymbol{y}|\boldsymbol{x})$ and contains two main scenarios, including code-to-code editing and comment&code-to-code editing.

For the code-to-code editing, our goal is to train a model $p(\boldsymbol{y}|\boldsymbol{x})$ that predicts the edited code $\boldsymbol{y}$ (*e.g.,* a new version) given the code to be edited $\boldsymbol{x}$ (*e.g.,* an old version). For the comment&code-to-code editing, our target is to train a model $p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{z})$ that predicts the edited code $\boldsymbol{y}$ given the code

[2]https://github.com/LJ2lijia/CodeEditor

to be edited $\boldsymbol{x}$ and a natural language comment $\boldsymbol{z}$. The comment describes code changes between $\boldsymbol{x}$ and $\boldsymbol{y}$.

2.2 Pre-trained Models for Source Code

Recently, pre-trained models for source code [24, 37] have been proposed and achieved SOTA results on many code-related tasks (*e.g.,* code generation [14, 16], code summarization [13]). Generally, pre-trained models consist of two stages - pre-training and fine-tuning.

During the pre-training, the models are trained with self-supervised pre-training tasks to learn prior knowledge about source code. Nowadays, code infilling tasks (*e.g.,* mask language modeling, MLM) [8, 37] have become the prevalent pre-training tasks for source code. Code infilling tasks randomly mask some tokens or spans in a program and ask the models to predict the masked content based on the contexts. In this way, code infilling tasks enable the models to learn the syntax of source code. Then, the pre-trained models are applied to specific tasks (*e.g.,* code editing) by fine-tuning with the task-specific data. Besides fine-tuning, pre-trained models can be applied in few-shot and zero-shot settings. The few-shot setting means that we fine-tune the pre-trained models with a few samples, and the zero-shot setting denotes the pre-trained models are used in specific tasks without training.

The performance of pre-trained models in downstream tasks mainly depends on the knowledge derived from the pre-training tasks. However, there is limited work exploring pre-training tasks specialized in code editing. In this paper, we propose a novel pre-training task for code editing and pre-train a code editing model named CodeEditor. We evaluate the pre-trained CodeEditor in fine-tuning, few-shot, and zero-shot settings.

2.3 Pre-trained Source Code Language Model

Language models aim to capture the statistical patterns in languages by assigning occurrence probabilities to a sequence of words. The models will score an utterance high, if it sounds "natural" to a native speaker, and score low the unnatural (or wrong) sentences. Programming languages are kinds of languages that contain predictable statistical properties and can be modeled by language models. Given a code token sequence $\boldsymbol{S} = \{s_1, s_2, ..., s_t\}$, the probability of this sequence is computed as:

$$p(S) = p\left(s_1\right) p\left(s_2 \mid s_1\right), \ldots, p\left(s_t \mid s_1 s_2, \ldots, s_{t-1}\right) \tag{1}$$

Recently, pre-trained language models (PTLMs) have shown to be effective [18, 20]. PTLMs are trained with a large-scale corpus of real code files. Given a partial code snippet, they can accurately predict multiple plausible patches. Thus, we utilize a powerful language model to rewrite an original program into a mutated version by replacing some original code spans with natural but inferior alternatives predicted by PTLMs. By editing the mutated code into its original version, the pre-trained model can know many diverse and meaningful edit patterns (*e.g.,* API updates, identifier renaming).

3 CODEEDITOR

3.1 Overview

The overview of our CodeEditor is shown in Figure 3. Our approach comprises three stages:

(1) Producing the pre-training data. We first collect lots of real-world programs from open-source communities (*e.g.,* GitHub). Then, we use a powerful generator to rewrite these programs into mutated versions. We view the mutated versions as the code to be edited and the corresponding original versions as the ground truth, to construct the pre-training data. The details are explained in Section 3.2.

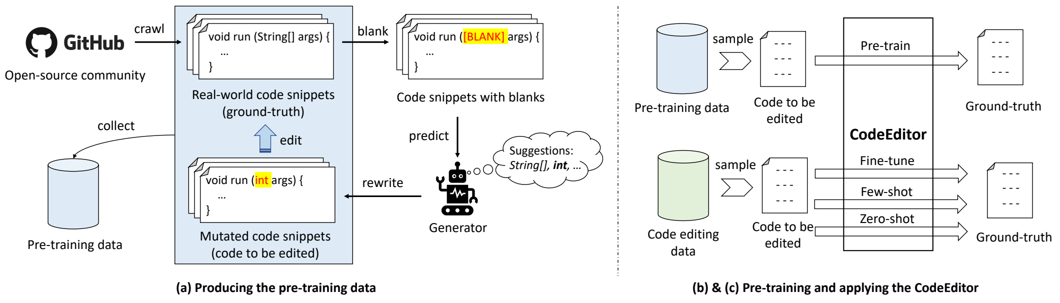

GitHub
Open-source community
crawl
void run (String[] args) {
...
}
Real-world code snippets (ground-truth)
blank
void run ([BLANK] args) {
...
}
Code snippets with blanks
collect
edit
predict
Suggestions: String[], int, ...
void run (int args) {
...
}
Mutated code snippets (code to be edited)
rewrite
Generator
Pre-training data
(a) Producing the pre-training data
Pre-training data
sample
Code to be edited
Pre-train
Ground-truth
CodeEditor
Fine-tune
Few-shot
Zero-shot
Code editing data
sample
Code to be edited
Ground-truth
(b) & (c) Pre-training and applying the CodeEditor

Fig. 3. The overview of CodeEditor.

(2) Pre-training the CodeEditor**.** We pre-train our CodeEditor with the pre-training data, which takes the code to be edited as inputs and outputs the edited code. The details are presented in Section 3.3.

(3) Applying the pre-trained CodeEditor**.** Finally, we apply the pre-trained CodeEditor to code editing in three settings - fine-tuning, few-shot, and zero-shot. The details are described in Section 3.4.

3.2 Producing the pre-training data

The goal of this step is to produce the pre-training data consisting of the code to be edited and the ground truth. As shown in Figure 3, we first collect lots of individual code snippets from open-source communities (*e.g.,* GitHub). These code snippets have been edited carefully and passed strict code reviews. Thus, we consider them as the ground truth. Then, for each code snippet $\boldsymbol{c}$, we utilize a strong generator to rewrite it into a mutated version $\boldsymbol{c'}$.

Given a code snippet $\boldsymbol{c} = [c_1, c_2, ..., c_n]$, c_i denotes i-th token and and n is the maximum length. We randomly select k spans in $\boldsymbol{c}$ and replace them with blanks. This process can be formulated to:

$$\begin{aligned} m_i &\sim \text{unif}\{1, n\} \text{ for } i = 1 \text{ to } k \\ l_i &\sim \text{unif}\{a, b\} \text{ for } i = 1 \text{ to } k \\ s_i &= [\boldsymbol{c}_{m_i}, ..., \boldsymbol{c}_{m_i+l_i}] \text{ for } i = 1 \text{ to } k \\ \boldsymbol{c}^{\text{blank}} &= \text{REPLACE}(\boldsymbol{c}, \boldsymbol{s}, \texttt{[BLANK]}) \end{aligned} \tag{2}$$

where $\boldsymbol{m} = [m_1, ..., m_k]$ and $\boldsymbol{l} = [l_1, ..., l_k]$ are the start positions and lengths of k spans, respectively. unif$\{a, b\}$ denotes a uniform distribution from a to b. $\boldsymbol{s} = [s_1, ..., s_k]$ is selected k spans. REPLACE$(\boldsymbol{c}, \boldsymbol{s}, \texttt{[BLANK]})$ is a function for replacing selected spans $\boldsymbol{s}$ in $\boldsymbol{c}$ with a specific token `[BLANK]`. As shown in Figure 3 (a), the token `String[]` is selected and replaced with `[BLANK]`.

A large-scale study of code edits [23] has found that the sizes of most repetitive edits range from 1 to 6 tokens. Therefore, the a and b in Equation 2 are set to 1 and 6, respectively. It means the length of each span is sampled from a uniform distribution of $\{1, 6\}$. Previous pre-trained work [39] also has shown that pre-trained models work better when 30% of tokens are modified. We follow this setting to control the number of spans k so that approximately 30% of code tokens are selected.

Next, we use a generator p_G to complete these blanks. For each blank, the generator will output many suggestions that are ranked based on probability. We consider the original span as the gold and pick the first non-gold suggestion as a sub-optimal alternative.

$$\begin{aligned} \hat{s}_i &\sim p_G\left(\boldsymbol{c}^{\text{blank}}\right) \text{ for } i = 1 \text{ to } k \\ \boldsymbol{c}' &= \text{REPLACE}(\boldsymbol{c}, \boldsymbol{s}, \hat{\boldsymbol{s}}) \end{aligned} \tag{3}$$

where $\hat{\boldsymbol{s}} = [\hat{s}_1, ..., \hat{s}_k]$ is selected sub-optimal alternatives predicted by the generator. Then, we replace original spans $\boldsymbol{s}$ with sub-optimal alternatives $\hat{\boldsymbol{s}}$ to obtain the mutated code $\boldsymbol{c}'$. In Figure 3 (a), the generator predicts multiple suggestions for the blank, *e.g.,* `String[]`, `int`. Thus, we select the first non-gold suggestion (*i.e.,* `int`) to complete the blank. Then, the original code snippet is rewritten into a mutated version that has a wrong parameter type.

By repeating the above process, we obtain lots of mutated versions and original versions of code snippets. Then, we consider the mutated versions as the code to be edited $\boldsymbol{x}$ and the corresponding original versions as the ground-truth $\boldsymbol{y}$, to construct our pre-training data. If the original code is equipped with a natural language comment z, we take the comment as an additional input, building a pre-training sample $(\boldsymbol{x}, z, \boldsymbol{y})$. Otherwise, the pre-training sample is built as $(\boldsymbol{x}, \boldsymbol{y})$. Our motivation is that these two types of samples correspond to two code editing scenarios. For samples without comments $(\boldsymbol{x}, \boldsymbol{y})$, CodeEditor learns a code-to-code edit ability that benefits the code-to-code editing scenario. For samples with comments $(\boldsymbol{x}, z, \boldsymbol{y})$, CodeEditor edits the source code with the guidance of comments, which is suitable for the code&comment-to-code editing scenario.

3.3 Pre-training the CodeEditor

As shown in Figure 3 (b), this step aims to pre-train CodeEditor with the pre-training data produced in Section 3.2. In this paper, we view the code editing task as a sequence-to-sequence task. Specifically, CodeEditor is trained to generate an output sequence Y based on an input sequence X. Following previous studies [3, 34], we build the input and output sequences as follows:

$$X = \begin{cases} \boldsymbol{x} & \text{without comments} \\ \boldsymbol{x}\texttt{[SEP]}z & \text{with comments} \end{cases} \qquad Y = \boldsymbol{y} \tag{4}$$

where `[SEP]` is a specific token for concatenating two sequences.

We train CodeEditor to generate the output sequence auto-regressively based on the input sequence by minimizing the following loss function:

$$\mathcal{L}(\theta) = -\log P(Y \mid X) = -\sum_{t=1}^{L} \log P(Y_t \mid X, Y_{<t}) \tag{5}$$

where Y_t denotes t-th token of the output sequence. θ denotes all trainable parameters and L is the maximum length of the output sequence.

3.4 Applying the pre-trained CodeEditor

The pre-trained CodeEditor learns a general code editing ability from the pre-training data. As shown in Figure 3 (c), the next step is to apply the pre-trained CodeEditor to code editing. In this work, we apply the pre-trained CodeEditor in three settings - fine-tuning, few-shot, and zero-shot settings. In the fine-tuning setting, we train the pre-trained CodeEditor with code editing datasets. In the few-shot setting, we sub-sample the datasets and train the pre-trained CodeEditor with the limited data. In the zero-shot setting, we directly apply the pre-trained CodeEditor to code editing applications without training.

For each setting, we apply the pre-trained CodeEditor to two code editing scenarios - code-to-code editing and comment&code-to-code editing. We follow the sequence-to-sequence learning and input-output representations used in the pre-training stage. For the code-to-code editing scenario, the input sequence is the code to be edited $\boldsymbol{x}$. For the comment&code-to-code editing scenario, we concatenate $\boldsymbol{x}$ and a natural language comment $\boldsymbol{z}$ into an input sequence. In both scenarios, the output sequence is the ground-truth $\boldsymbol{y}$. Then, we train the pre-trained CodeEditor to generate an output sequence based on an input sequence by minimizing the loss function (Equation 5).

3.5 Model Architecture

CodeEditor is an encoder-decoder neural network based on Transformer [36], which contains a bidirectional encoder and an auto-regressive decoder. Following previous studies [37], we adopt the same architecture as Text-to-Text-Transfer Transformer (T5) model [27]. (1) The number of layers (*i.e.,* Transformer blocks) L = 6. (2) The dimension of the model d_{model} = 512. (3) The dimension of feed-forward layers d_{ff} = 2048. (4) The number of self-attention heads h = 8 and the drop rate p=0.1. The total number of parameters is 60M.

4 STUDY DESIGN

To evaluate the effectiveness of our CodeEditor, we design three research questions (RQ) and perform a large-scale study to answer these questions. In this section, we describe the research questions and details of our study.

4.1 Research Questions

We argue that our pre-training task can improve the performance of code editing models. To prove the effectiveness of our CodeEditor, in the RQ1 and RQ2, we fine-tune the pre-trained CodeEditor in two code editing scenarios and compare it to SOTA baselines.

RQ1: How does CodeEditor **perform in the code-to-code editing scenario?**

In this RQ, we fine-tune the pre-trained CodeEditor with two code-to-code editing datasets, respectively. Given a code snippet to be edited, CodeEditor is trained to generate a new version by automating generic code changes. Then, we evaluate the performance of CodeEditor on the test data and compare it with SOTA baselines on two datasets.

RQ2: How does CodeEditor **perform in the comment&code-to-code editing scenario?**

In this RQ, we fine-tune the pre-trained CodeEditor with two comment&code-to-code editing datasets, respectively. Given a code snippet to be edited and a natural language comment, the model outputs a new version by automating code changes described by the comment. We also compare CodeEditor with SOTA baselines on the test data.

We hypothesize that our pre-training task endows the model with a practical code editing ability and strengthens the model's generalization ability. To prove this point, we design the RQ3 that evaluates the pre-trained CodeEditor in few-shot and zero-shot settings where the fine-tuning data is limited.

RQ3: How does CodeEditor **perform in few-shot and zero-shot settings?**

For the zero-shot setting, we evaluate the pre-trained CodeEditor on the test data without training. For the few-shot setting, we sub-sample (*i.e.,* 10%, 20%, 30%, 40%, 50%) code editing datasets. We fine-tune the pre-trained CodeEditor with the limited data and measure its performance on the test data.

4.2 Datasets

Our CodeEditor involves two steps - pre-training and fine-tuning. For the pre-training, we need a large-scale corpus consisting of real-world code snippets for producing the pre-training data. For the fine-tuning, we require the code editing datasets containing real-world edit pairs for transferring the pre-trained CodeEditor.

Table 1. Pre-training and fine-tuning datasets (# instances)

Dataset	train	dev	test
Pre-training			
CodeSearchNet [11]	1,519,885	50,000	-
Code-to-code editing			
Small [33]	46,628	5,828	5,831
Medium [33]	52,324	6,542	6,538
Comment&code-to-code editing			
Tufano et al. [35]	13,670	1,713	1,704
new_large [34]	134,209	16,773	16,780

4.2.1 Pre-training. Following previous pre-trained studies [8, 10, 24, 37], we select the CodeSearchNet-java [11], which is collected from the popular open-source repositories from GitHub. The CodeSearchNet-java dataset contains 1.5 million individual Java methods, including 499,618 methods with comments and 1,070,267 methods without comments. For each method, we follow the pipeline in Section 3.2 and obtain a mutated version. Then, we consider the modified methods as an input and the original methods as an output, to construct a pre-training sample.

We split the pre-training data into a train set and a dev set. The data statistics are shown in Table 1. To ensure a fair comparison with previous studies, we do not use additional data for pre-training.

4.2.2 Fine-tuning. In our experiments, we apply the pre-trained CodeEditor to two code editing scenarios, *i.e.,* code-to-code editing, and comment&code-to-code editing. For each scenario, we select two public datasets for fine-tuning.

Code-to-code editing. We use two Java code editing datasets proposed by Tufano et al. [33]. Tufano et al. extracted method-level pairs from commits in GitHub repositories. Each pair is composed of a previous version and a new version of a Java method. Based on the length of extracted methods, Tufano et al. provided two datasets: Small and Medium datasets, with the former having a code length below 50 tokens and the latter having a code length between 50-100 tokens. Note that we use the raw version of two datasets. The data statistics are shown in Table 1.

Comment&code-to-code editing. We use two Java code editing datasets proposed by Tufano et al. [34, 35]. The datasets are collected from Java open-source projects on GitHub. Tufano et al. further filtered low-quality projects and extracted triplets as samples. Each triplet contains a previous version and a new version of a Java method, and a natural language comment that describes code changes. The data statistics are shown in Table 1.

4.3 Evaluation Metrics

Following previous code editing studies [4, 27, 32], we use the *exact match*, *CrystalBLEU*, and *Edit distance* as evaluation metrics. We regard the code edited by models as a prediction and the code edited by human developers as the ground truth.

- **Exact match (EM)** is the percentage of predictions that has the same token sequence as the ground truth.
- **CrystalBLEU** [7] is a metric to precisely and efficiently evaluate code similarity despite trivially shared n-grams. CrystalBLEU is an extension of BLEU [25] that removes trivially shared n-grams before computing the n-gram overlap between two pieces of code. Extensive experiments show that CrystalBLEU provides higher distinguishability than standard BLEU

on the code. We reuse official implementations of CrystalBLEU and use training sets of experimental datasets to extract trivially shared n-grams.
- **Edit distance** is the minimum number of token edits (insertions, deletions, or substitutions) needed to convert a prediction into the ground truth. The lower the Levenshtein distance, the closer the prediction is to the ground truth.

4.4 Baselines

We select 13 recently proposed code editing models as baselines. They can be divided into two categories: baselines trained from scratch and pre-trained baselines.

Baselines trained from scratch denote that baselines are randomly initialized and further trained with code editing datasets.

- **Tufano et al. [33]** is a pioneer work that utilizes a DL-based model to learn code changes from pull requests.
- **Tufano et al. [35]** proposes two DL-based code editing models that can automate generic code changes and changes recommended by reviewers' comments.
- **CODIT [3]** is a tree-based code editing model that leverages the rich syntactic structure of code and generates syntactically correct changes.
- **Transformer [36]** is a popular DL-based model and has obtained promising results in code-related tasks.
- **AutoTransform [32]** is a variant of Transformer. It introduces a Byte-Pair Encoding (BPE) algorithm [29] to handle unseen tokens and utilizes the self-attention mechanism to model long sequences.

Pre-trained baselines are firstly pre-trained with several pre-training tasks and then fine-tuned with the code editing task. Nowadays, pre-trained code editing models have achieved SOTA results on many benchmarks.

- **RoBERT (code) [19]** is a pioneer pre-trained model for natural languages. We continually pre-train it with the source code for comparison.
- **CodeBERT [8]** is a classic pre-trained model for source code. It applies the pre-training tasks for natural languages to the source code and has been widely used in code generation.
- **GraphCodeBERT [10]** is a augmented variant of CodeBERT. It proposes two new pre-training tasks to learn the relationships between the data flow graph and code tokens.
- **CodeGPT [20]** is a variant of GPT-2 [26] that is a powerful pre-trained model for natural language generation. CodeGPT is initialized with GPT-2 and continually pre-trained with the source code.
- **SPT-Code [24]** is a sequence-to-sequence pre-trained model for source code. It utilizes three pre-training tasks to learn the lexical and syntactic knowledge of source code.
- **T5-review [34]** applies a popular pre-trained model - T5 [27] for natural languages into source code and has obtained promising results in code editing.
- **MODIT [4]** is designed for the comment&code-to-code editing. MODIT considers commit messages as intents for guiding code editing models.
- **CodeT5 [37]** is a recently proposed pre-trained model for source code. CodeT5 employs identifier-aware pre-training tasks and has achieved SOTA results on many code-related tasks.

Note that some baselines (*i.e.,* Tufano et al. [33], CODIT, and AutoTransform) are designed for the code-to-code editing scenario, and MODIT only works in the comment&code-to-code editing scenario. Thus, we compare our CodeEditor to these special baselines in specific scenarios. Besides, considering that some pre-trained baselines (*i.e.,* RoBERTa, CodeBERT, and GraphCodeBERT) only contain an encoder, we follow previous work [4] and add a six-layer transformer decoder along with these baselines to support code editing. The decoder is randomly initialized and optimized during fine-tuning.

4.5 Implementation Details

4.5.1 Pre-training Details. We implement our CodeEditor with a deep learning development framework - Pytorch[3] and a python package - transformers[4]. We use Byte-Pair Encoding (BPE) [29] to tokenize the source code and natural language comments into tokens. The network architecture of CodeEditor is described in Section 3.5. We initialize our model with the pre-trained CodeT5-small [37] (60M), and continually pre-train the model with our pre-training task. Note that initializing with existing pre-trained weights is common in previous studies [4, 10, 20]. To make a fair comparison, we also reuse the pre-trained CodeT5-small. This also reduces recent concerns [30] about the carbon footprint and energy consumption caused by the pre-training from scratch. We pre-train our CodeEditor on a cluster of 32 NVIDIA A100 GPUs with 40G memory. We set the maximum input and output sequence lengths to 512. We pre-train the model for 180,000 steps, with a batch size of 512, and a learning rate of 5e-5 with a linear warm-up for the first 10,000 steps. Every 10,000 pre-training steps, we measure the performance of our model by calculating the loss of our model on the dev set, but without updating the parameters. Finally, we select the checkpoint with a minimum loss on the dev set as the pre-trained CodeEditor.

4.5.2 Fine-tuning Details. During the fine-tuning step, we set the batch size to 32 and the learning rate to 5e-5. The warmup steps are 1000 steps. During testing, we use the beam search and set the beam size to 10.

5 RESULTS AND ANALYSES

In this section, we answer three research questions (Section 4.1) based on our experimental results.

5.1 Performance in the code-to-code editing scenario

RQ1: How does CodeEditor perform in the code-to-code editing scenario?

Motivation. We investigate whether our CodeEditor learns a strong code editing ability to support the code-to-code editing application.

Setup. We fine-tune or train baselines and the pre-trained CodeEditor with two code-to-code editing datasets (*i.e.*, Small and Medium) described in Section 4.2. Then, we use metrics presented in Section 4.3 to evaluate the performance of baselines and our CodeEditor on the test data. Since MODIT [4] only works in the comment&code-to-code editing scenario, we omit it in this experiment.

Results and Analyses. The experimental results in two code-to-code editing datasets are shown in Table 2. The values in parentheses are relative improvements compared to the SOTA baseline. We have three-fold findings. (1) Our CodeEditor outperforms all baselines on both datasets. Specifically, CodeEditor generates more correct programs than the SOTA baseline - CodeT5 by 15% in the Small dataset and 25.5% in the Medium dataset. Note that exact match is a strict metric and is difficult to be improved. The significant improvements show the superiority of our CodeEditor in automatic code editing. Moreover, CodeEditor obtains better results on the CrystalBLEU and Edit distance. It shows that for incorrect predictions, the outputs of CodeEditor are closer to the ground truth and cost fewer developers' efforts to edit. We attribute these improvements to our pre-training task. Compared to previous code infilling tasks, we introduce

[3]https://pytorch.org/

[4]https://huggingface.co/docs/transformers/index

Table 2. The performance of baselines and our CodeEditor on two code-to-code editing datasets. The values in parentheses are relative improvements compared to the SOTA baseline.

Type	Approaches	Small dataset			Medium dataset		
		Exact match ↑	CrystalBLEU ↑	Edit distance ↓	Exact match ↑	CrystalBLEU ↑	Edit distance ↓
Trained from Scratch	Tufano et al. [33]	3.45	63.36	31.11	1.62	76.23	28.53
	Tufano et al. [35]	3.91	63.95	30.86	1.84	76.71	27.79
	CODIT	6.53	65.82	22.40	3.27	75.58	25.75
	Transformer	7.01	68.40	25.21	4.19	78.73	27.65
	AutoTransform	7.65	70.24	24.20	4.51	79.93	25.97
Pre-trained Model	RoBERTa (code)	13.75	76.48	17.50	4.67	82.62	23.45
	CodeBERT	14.70	77.21	15.18	4.97	84.01	22.64
	GraphCodeBERT	15.91	78.34	15.30	7.49	84.49	22.48
	CodeGPT	15.09	77.69	15.08	5.29	83.07	26.07
	T5-review	14.82	78.99	14.94	5.05	83.59	24.00
	SPT-Code	17.54	80.25	14.99	9.39	87.62	21.51
	CodeT5	20.36	80.94	14.62	10.03	87.51	20.70
	CodeEditor	**23.41 (↑ 15%)**	**83.11**	**13.12**	**12.59 (↑ 25.5%)**	**91.54**	**17.59**

a powerful language model to derive more meaningful edit patterns (*e.g.,* type/object changes, API updates) that probably occur in real-world code editing. These edit patterns help our CodeEditor learn the prior knowledge of code editing and strengthen the code editing ability in practical applications. (2) Pre-trained models are more promising. Compared to models trained from scratch, pre-trained models achieve significant improvements. For example, in terms of the exact match, our CodeEditor improves the Transformer by 234% in the Small dataset and 200.5% in the Medium dataset, despite both models having the same architecture and a comparable number of parameters. The improvements show that pre-training tasks are necessary and effective for code editing. (3) Sequence-to-sequence (Seq2Seq) learning is beneficial to code editing. We notice that Seq2Seq-based pre-trained models (*e.g.,*CodeEditor, CodeT5) obtain better results compared with encoder-only (*e.g.,* CodeBERT) and decoder-only (*e.g.,* CodeGPT) pre-trained models. This is because both understanding the code to be edited and correctly generating the edited code are important for code editing. The Seq2Seq pre-trained model is a better choice that contains a pre-trained encoder for understanding the code and a pre-trained decoder for generating the code.

Answer to RQ1: After being fine-tuned with two code-to-code editing datasets, CodeEditor achieves the best results among all baselines. In particular, CodeEditor generates 23.41% and 12.59% correct programs on two datasets, outperforming the SOTA baseline by 15% and 25.5%. The significant improvements show the effectiveness of our CodeEditor in automatic code-to-code editing.

5.2 Performance in the comment&code-to-code scenario

RQ2: How does CodeEditor perform in the comment&code-to-code editing scenario?

Motivation. We investigate whether our CodeEditor learns to edit source code under the guidance of a natural language comment.

Setup. We fine-tune or train baselines and the pre-trained CodeEditor with two comment&code-to-code editing datasets (*i.e.,* Tufano et al. [35] and new_large). Then, we use three metrics (*i.e.,* exact match, CrystalBLEU, and Edit distance) to evaluate the performance of baselines and our CodeEditor on the test data. Since some baselines (*i.e.,* Tufano et al. [33], CODIT, and AutoTansform) are designed for code-to-code editing, we omit them in this experiment.

Results and Analyses. The experimental results are shown in Table 3. The values in parentheses are relative improvements compared to the SOTA baseline. We obtain two-fold insights. (1) Our

Table 3. The performance of baselines and our CodeEditor on two comment&code-to-code editing datasets. The values in parentheses are relative improvements compared to the SOTA baseline.

Type	Approaches	Tufano et al. [35] dataset			new_large dataset		
		Exact match ↑	CrystalBLEU ↑	Edit distance ↓	Exact match ↑	CrystalBLEU ↑	Edit distance ↓
Trained from Scratch	Tufano et al. [35]	2.93	49.57	36.70	0.89	47.87	63.44
	Transformer	7.63	52.13	33.26	1.31	51.90	59.06
Pre-trained Model	RoBERTa (code)	13.44	77.24	20.83	4.60	76.11	46.88
	CodeBERT	13.79	79.69	19.11	4.89	77.50	45.83
	GraphCodeBERT	21.36	81.29	17.69	7.73	80.54	42.97
	CodeGPT	14.96	82.13	18.77	4.96	80.05	44.36
	T5-review	15.26	83.42	17.28	5.02	81.73	42.64
	SPT-Code	23.30	84.79	14.32	7.78	82.63	41.39
	CodeT5	26.88	85.42	13.75	7.81	83.32	39.02
	CodeEditor	**29.40 (↑ 9.4%)**	**87.40**	**11.12**	**9.89 (↑ 26.6%)**	**86.46**	**36.74**

CodeEditor obtains the best results among all baselines. Specifically, in terms of exact match, our CodeEditor outperforms the SOTA baseline - CodeT5 by 9.4% in the Tufano et al. [35] and 26.6% in the new_large dataset. We attribute these improvements to our pre-training data that contains edit pairs with comments. These edit pairs ask our model to edit the input code into a new version that is consistent with a given comment. Thus, our model learns a conditional code editing ability and performs well in comment&code-to-code editing. (2) Understanding natural language comments is crucial to the comment&code-to-code editing. Compared to code-to-code editing, comment&code-to-code editing is a more challenging task that requires understanding the semantics of given natural language comments. Table 3 shows that models trained from scratch have poor results and pre-trained models perform well. The reason is that models trained from scratch lack prior knowledge about the source code and natural languages. Thus, it is hard for models trained from scratch to understand the semantics of comments through the limited data. While pre-trained models are pre-trained with extensive code files containing natural language comments. They can learn related knowledge about the source code and natural languages, thus obtaining significant improvements.

Answer to RQ2: After being fine-tuned with two comment&code-to-code editing datasets, CodeEditor generates 29.4% and 9.89% correct programs, improving the SOTA baseline by 9.4% and 26.6%. It shows the CodeEditor learns a high-level natural language understanding ability and a strong conditional code editing ability.

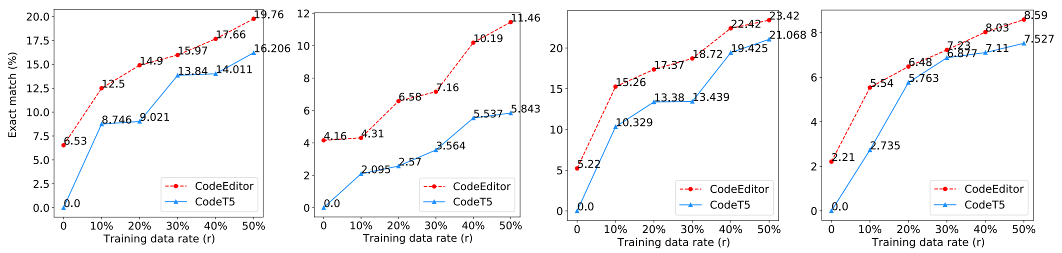
5.3 Performance in few-shot and zero-shot settings

RQ3: How does CodeEditor perform in few-shot and zero-shot settings?

Motivation. We think our pre-training task is closer to real-world code editing and strengthens the model's generalization ability. Thus, we evaluate the performance of our CodeEditor in few-shot and zero-shot settings where the fine-tuning data is limited.

Setup. For the few-shot setting, we sub-sampling the code editing datasets. We fine-tune the pre-trained CodeEditor with these limited datasets and measure its performance. For the zero-shot setting, we evaluate the pre-trained CodeEditor without fine-tuning. We conduct zero-shot and few-shot experiments on four datasets used in RQ1 and RQ2. To prove the superiority of our pre-training technique, we compare our CodeEditor to the SOTA pre-trained baseline - CodeT5 in terms of exact match.

Specifically, we select r (%) samples from a code editing dataset for fine-tuning and then evaluate fine-tuned models on the entire test data. When $r = 0$, the model is not fine-tuned and evaluated



(a) Small dataset (b) Medium dataset (c) Tufano et al. [22] dataset (d) new_large dataset

Fig. 4. A comparison of CodeT5 and CodeEditor in zero-shot and few-shot learning.

```
public void load(String filename){
-    reader = new BufferedReader(filename);
+    reader = new BufferedReader(new FileReader(filename));
}
```

(a) A pre-training sample in our work

```
public boolean isXml(File file)
-    scanner = new Scanner(new BufferedReader(file));
+    scanner = new Scanner(new BufferedReader(
                                      new FileReader(file)));
}
```

(b) A real-world sample in the Small dataset

Fig. 5. Similar edits in our pre-training data and real-world code editing data.

on the entire test data (*i.e.,* zero-shot learning). When $r > 0$, the model is fine-tuned with only a few samples and then tested on the entire test data. In our experiments, r is set to 0%, 10%, 20%, 30%, 40%, 50%.

Results and Analyses. The experimental results on four datasets are shown in Figure 4. We have two insights. (1) In the zero-shot setting, our pre-trained CodeEditor successfully edits some real-world code snippets while CodeT5 can not work. Specifically, in the zero-shot setting ($r = 0$), our pre-trained CodeEditor correctly generates 6.53%, 4.16%, 5.22%, and 2.21% programs, for a total of 1,113 programs on four datasets. While none of the generated programs from CodeT5 is correct. This observation validates that existing code infilling-based pre-trained models have limitations in code editing. In this paper, we propose a novel pre-training task that is closer to real-world code editing. Figure 5 shows our pre-training example and a real-world example from the Small dataset. We can see that both samples share an edit pattern. Thus, the edit patterns learned from our pre-training task can be seamlessly applied to practical applications without fine-tuning. (2) In few-shot learning, our pre-trained CodeEditor substantially outperforms the CodeT5. As the data size increases, the performance of CodeEditor grows steadily and substantially outperforms CodeT5 on four datasets. We also notice that CodeEditor fine-tuned with a few samples even outperforms several baselines fine-tuned with all samples. For example, when being fine-tuned with 50% samples, CodeEditor outperforms baselines that fine-tuned with all samples (*i.e.,*CodeEditor: 19.76% *vs.* SPT-Code: 17.54% on the Small dataset, CodeEditor: 11.46% *vs.* CodeT5: 10.03% on the Medium dataset, CodeEditor: 23.42% *vs.* SPT-Code: 23.30% on the Tufano et al. [35] dataset, CodeEditor: 8.59% *vs.* CodeT5: 7.81% on the new_large dataset). The results show that compared to previous pre-trained models, our pre-trained CodeEditor shows a strong generalization ability in code editing.

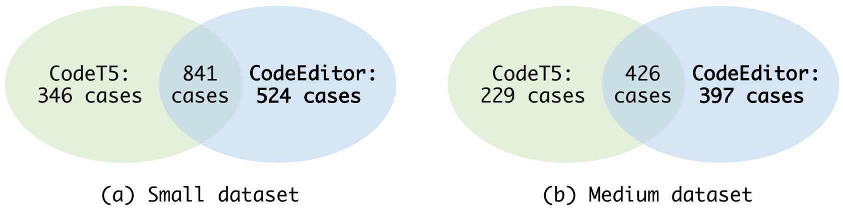


Fig. 6. The statistics of successful cases on the Small and Medium datasets.

```
Natural language comment: requireNonNull is not needed given copyOf checks for nullability.
Code to be edited:
public ValuesMatcher(Map outputSymbolAliases, Integer expectedOutputSymbolCount, List expectedRows){
    this.outputSymbolAliases = ImmutableMap.copyOf(requireNonNull(outputSymbolAliases));
    this.expectedOutputSymbolCount = requireNonNull(expectedOutputSymbolCount);
    this.expectedRows = requireNonNull(expectedRows);
}
-----------------------------------------------------------------------------------------------
SPT-Code & T5-review:
public ValuesMatcher(Map outputSymbolAliases, Integer expectedOutputSymbolCount, List expectedRows){
    this.outputSymbolAliases = requireNonNull(outputSymbolAliases);
    . . .
}
-----------------------------------------------------------------------------------------------
CodeT5:
public ValuesMatcher(Map outputSymbolAliases, Integer expectedOutputSymbolCount, List expectedRows){
    this.outputSymbolAliases = ImmutableMap.copyOf(outputSymbolAliases);
    this.expectedOutputSymbolCount = expectedOutputSymbolCount;
    this.expectedRows = expectedRows;
}
-----------------------------------------------------------------------------------------------
CodeEditor & Ground truth:
public ValuesMatcher(Map outputSymbolAliases, Integer expectedOutputSymbolCount, List expectedRows){
    this.outputSymbolAliases = ImmutableMap.copyOf(outputSymbolAliases);
    . . .
}
```

Fig. 7. A successful example on the Tufano et al. [35] dataset. To make it clear, we underline edited contents and omit some unchanged statements.

Answer to RQ3: In the zero-shot setting (*i.e.,* without fine-tuning), CodeEditor correctly edits 1,113 programs while CodeT5 can not work. In the few-shot setting (*i.e.,* being fine-tuned with a few samples), CodeEditor substantially outperforms CodeT5 and some baselines fine-tuned with all samples. It shows our pre-training task is closer to real-world code editing and strengthens the model's generalization ability.

6 DISCUSSION

6.1 Case study

Case statistics. We collect successful cases of CodeT5 and our CodeEditor on the Small and Medium datasets. Figure 6 shows the statistics of successful cases. We can see that CodeT5 and CodeEditor both can correctly solve some samples. Compared to CodeT5, CodeEditor solves unique 524 samples and 397 samples on two datasets, respectively. This result shows that CodeEditor is complementary to existing SOTA models for code editing.

```
public void run(){
   setCanMove(true);
   reloadtime.cancel();
}
```

(a) Input code

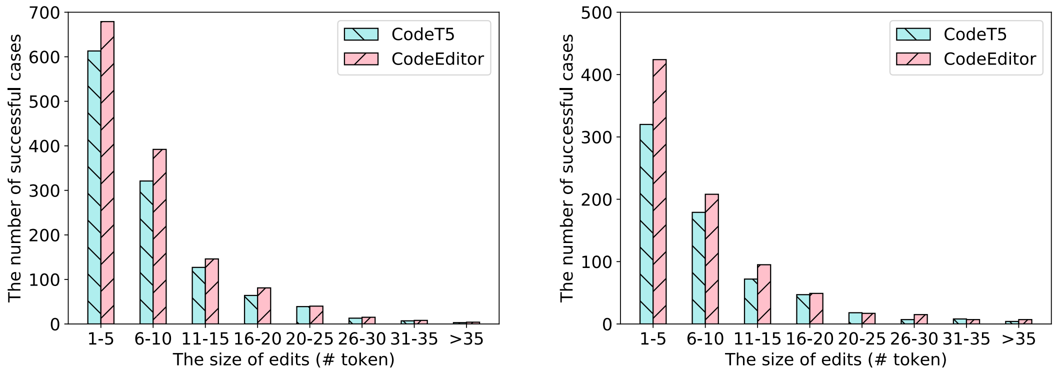
```
public void run(){
   setCanMove(true);
}
```

(b) Ground-truth

```
public void run(){
   setCanMove(true);
   reloadtime.cancel(true);
}
```

(c) Generated code

Fig. 8. A failed example of CodeEditor on the Small dataset.



(a) Small dataset (b) Medium dataset

Fig. 9. The performance on different edit sizes.

Successful case. Figure 7 presents a successful example of our CodeEditor on the Tufano et al. [35] dataset. In this example, the natural language comment suggests removing an unnecessary call (*i.e.,* `requireNonNull`) in line 2. However, SPT-Code and T5-review wrongly delete the API `ImmutableMap.copyOf`, and CodeT5 mistakenly removes all `requireNonNull` calls. This is because previous pre-trained models are pre-trained to infill a given blank instead of automatically determining edit locations based on comments. Thus, baselines often misunderstand the comment and predict wrong edits. While our CodeEditor is pre-trained to understand the semantics comments and edit the code with the guidance of comments. Therefore, CodeEditor can accurately locate the line to be edited and conduct a correct edit, without introducing other defects.

Failed case. Figure 8 shows a failed example of CodeEditor on the Small dataset. In this example, the expected edit is to remove a redundant statement. While our CodeEditor mistakenly modifies this statement by adding an argument (*i.e.,* `true`). We speculate that there are two reasons for the failed example. ❶ CodeEditor does not how to edit. The edit in this example is rare in the data. Without additional guidance (*e.g.,* a comment), CodeEditor has no idea how to edit. It also shows that a natural language comment is important to perform code editing. ❷ CodeEditor tends to modify statements instead of deleting statements. As stated in Section 3.3, we randomly rewrite some spans of an original program to build the pre-training data. The lengths of spans are in a range of [1, 6], and the selected spans mainly are parts of statements. Thus, our pre-training data mainly is to modify statements and contains a small number of deleting complete statements. Therefore, the pre-trained CodeEditor tends to modify statements and is relatively weak in deleting complete statements. To address this problem, we will observe real-world code edits (*e.g.,* the sizes and types) and construct more effective pre-training data in future work.

Table 4. The comparison (exact match) of CodeT5-cont and our CodeEditor.

Approaches	Code-to-code editing		Comment&code-to-code editing	
	Small	Medium	Tufano et al. [35]	new_large
CodeT5	20.36	10.03	26.88	7.81
CodeT5-cont	20.40	10.08	25.53	8.32
CodeEditor	**23.41**	**12.59**	**29.40**	**9.89**

6.2 Performance on different edit sizes

We also analyze the performance of different approaches on different (token-level) edit sizes. Specifically, we divide test data into multiple groups based on the edit sizes, such as one-token edits and two-token edits. For each group, we collect the outputs of different models and report the number of successful cases. The experimental results are shown in Figure 9. We can see that our CodeEditor outperforms the SOTA baseline - CodeT5 on different edit sizes. In particular, on edits of less than 10 tokens, CodeEditor keeps stable and prominent improvements over CodeT5. The results show the superiority and robustness of our model. As the edit size increase, the number of test samples decreases and the improvements of our CodeEditor are relatively slight.

6.3 Influence of more training steps

Following previous studies [10, 17, 20], we initialize our model with the weights of CodeT5, and continually pre-train the model with our pre-training task. Therefore, there is a question of whether more pre-training steps give our model an unfair advantage. To address this problem, we also continually pre-train CodeT5 using its original pre-training tasks and the number of training steps is equal to that of our model. We mark the continually pre-trained CodeT5 as CodeT5-cont.

The comparison of CodeT5-cont and our CodeEditor is shown in Table 4. Continuing pre-training brings CodeT5-cont negligible improvements and even degrades the performance on the Tufano et al. [35] dataset. While our CodeEditor outperforms CodeT5-cont by a substantial margin on four datasets. Therefore, we can conclude that CodeEditor is still superior under the same amount of pre-training steps and data. In addition, the results demonstrate that code infilling tasks provide limited help to code editing. Our pre-training task is closer to the real-world code editing and can provide stronger supervision signals. Thus, our CodeEditor significantly improves CodeT5-cont.

6.4 Threats to validity

Threats to external validity relate to the quality of experimental datasets and the generalizability of our results. First, to ensure fairness of the comparison, we follow previous studies [8, 10, 37] and use CodeSearchNet for pre-training. The four code editing datasets for fine-tuning are mainstream benchmarks and have been used in many related works [4, 33–35]. For rigorous consideration, we remove all duplicate samples between pre-training data and test data to avoid data leakage. Second, we only conduct experiments on the Java datasets. Although Java may not be representative of all programming languages, we conduct experiments on four datasets that are large and safe enough to show the effectiveness of our model. Besides, our model uses only language-agnostic features and can be applied to other programming languages.

Threats to internal validity include the influence of hyper-parameters. It is widely known that deep learning models are sensitive to hyper-parameters. For the baselines, we use the source code provided by their original papers and ensure that the model's performance is comparable with their reported results. For our CodeEditor, due to the high training cost of pre-training, we do a small-range grid search on the learning rate and batch size, leaving other hyper-parameters the same as those in previous studies [37]. Previous work [37] has explored effective settings of hyper-parameters through extensive experiments. Thus, there may be room to tune more hyper-parameters for more improvements.

Threats to construct validity relate to the reliability of our evaluation metrics. To address this threat, we use the *exact match*, *CrystalBLEU* score and *Edit distance* as evaluation metrics. The exact match evaluates the percentage of correctly predicted code snippets. It is a mainstream metric for code editing and is used in almost all previous studies [4, 15, 33–35]. Considering exact match may be too strict, we further employ CrystalBLEU and Edit distance to measure the text-similarity between predictions and the ground truth. Based on the above metrics, each experiment is run three times and the average result is reported.

7 RELATED WORK

In this paper, we propose a new pre-training model for automatic code editing. Thus, our work mainly relates to two research areas: (i) pre-training for source code, and (ii) code editing. In this section, we summarize related work in these two areas.

7.1 Pre-training for Source Code

Recently, some pre-trained models for source code have been proposed and applied to a variety of software engineering (SE) tasks, such as code search [8, 10] and code summarization [24, 37]. Previous studies first pre-train a model with pre-training tasks, then fine-tune the pre-trained model with specific tasks. The prior knowledge learned from pre-training tasks can enhance the performance and generalization ability of models in downstream tasks. According to the network architecture, existing pre-trained models for source code can be categorized into three groups: encoder-only, decoder-only, and Seq2Seq-based models.

Encoder-only pre-trained models only contain an encoder and are trained to learn a generic code representation for code understanding tasks (*e.g.,* code classification). CodeBERT [8] is a pioneer encoder-only model that reuses previous pre-training tasks in the NLP filed [19]. The obvious improvements by CodeBERT prove the effectiveness of pre-training techniques. Guo et al. [10] further introduced data flow graphs into pre-trained models to help models understand the inherent structure of source code.

Decoder-only pre-trained models consist of a decoder and are mainly used for code generation tasks (*e.g.,* code completion). Liu et al. [18] proposed a multi-task-based pre-trained model and applied it to code completion. Inspired by the progress made by GPT models in the NLP field, some researchers [20] transferred the GPT-2 [26] to source code and proposed a CodeGPT model, which achieved SOTA results on the code completion task.

Seq2Seq-based pre-trained models consist of an encoder and a decoder and can support code understanding and generation tasks. Early studies mainly followed ideas (*e.g.,* T5 [27], BART [12]) for natural languages and applied them into source code, such as T5-learning [21], PLBART [1]. Recently, Niu et al. [24] designed three pre-training tasks for source code and proposed a novel pre-trained model named SPT-Code. Wang et al. [37] proposed a pre-trained model named CodeT5 that better leverages the code semantics conveyed from the developer-assigned identifiers. Nowadays, CodeT5 has become the SOTA model for various code-related tasks.

Differences. Although the above pre-trained models can be applied to code editing, their pre-training tasks are not designed for code editing and can be further improved. This paper aims to propose a novel pre-training task for code editing. Our pre-training task trains a model to edit an auto-rewritten mutated code snippet into the ground truth, to learn edit patterns. Experimental results show that our CodeEditor outperforms existing code editing baselines in three settings (*i.e.,* fine-tuning, few-shot, and zero-shot).

We notice that some similar studies (*i.e.,* ELECTRA [5] and SSR [39]) use a generator to rewrite original natural language sentences into a new version and construct the pre-training data. We think the differences between our work and these approaches are three-fold. ❶ ELECTRA [5] only rewrites single tokens in original samples and trains a model to distinguish which tokens are modified. While our CodeEditor rewrites several spans of different lengths in original samples and trains a model to reconstruct rewritten spans. Thus, compared to ELECTRA, CodeEditor can learn more edit patterns and is closer to code editing. ❷ SSR [39] randomly rewrites several spans of original samples and uses specific tokens (<s>, </s>) to mark the locations of rewritten spans. For example, a pre-training input of SSR is `In 2001, Elon Musk joined SpaceX, <s> a manufacturer </s> company`. Then, SSR is trained to edit rewritten spans and output original spans (*e.g.,* `an aerospace manufacturer`). While our CodeEditor does not specify locations of rewritten spans. Our CodeEditor is trained to automatically locate parts that need to be edited and outputs patches. Our motivation is that code editing does not provide locations in practice. Compared to SSR, CodeEditor is closer to practical scenarios and performs better in real-world code editing data. ❸ ELECTRA [5] and SSR [39] may produce trivial pre-training samples, *i.e.,* the mutated sample is the same as the original sample. It teaches models few edit patterns and even misleads models to copy inputs as outputs. While our CodeEditor ensures mutated samples are different from original samples. By learning from our produced data, CodeEditor knows many meaningful and diverse edit patterns.

7.2 Code Editing

The motivation of code editing is to model edit patterns in previous code edits and apply these patterns to newly-written code snippets. It can be framed as a sequence-to-sequence task in which the code to be edited is transformed into the edited code. Nowadays, code editing mainly contains two scenarios, including code-to-code editing and comment&code-to-code editing.

Code-to-code editing aims to edit a code snippet into a new version by automating generic code changes. Tufano et al. [33, 35] presented an initial investigation of using deep neural networks in editing the source code. They collected code changes from pull requests and proposed a standard Seq2Seq model to learn code changes. Chakraborty et al. [3] further proposed a tree-based model that learned to edit the syntax tree of source code and ensured the grammatical correctness of the edited code. Thongtanunam et al. [32] introduced the BPE algorithm and the Transformer architecture to handle new tokens (*e.g.,* rare identifiers) and too-long code snippets. Recently, some powerful pre-trained models are applied to code editing and achieved SOTA results, such as SPT-Code [24], T5-review [34], and CodeT5 [37].

Comment&code-to-code editing aims to edit a code snippet by automating code changes that are recommended by a natural language comment. Compared to code-to-code editing, comment&code-to-code editing requires understanding the natural language comments and is a challenging task. Tufano et al. [35] collected edit pairs from the code review and proposed a Seq2Seq model to edit developers' programs based on reviewers' comments. Inspired by the pre-training techniques, Tufano et al. [34] applied ideas [27] in the NLP field into the source code and proposed a pre-trained code editing model named T5-review. They also released a new dataset named new_large containing 130,000 edit pairs with comments. Besides, some researchers tried to apply existing pre-trained models to code editing. Chakraborty et al. [4] considered the commit message as the guidance and utilized a pre-trained model - PLBART to edit programs.

8 CONCLUSION AND FUTURE WORK

In this paper, we propose a novel pre-training task for code editing and present an effective pre-training code editing model named CodeEditor. Compared to previous approaches, our pre-training task endows CodeEditor with a more effective code editing ability and a stronger generalization ability. Specifically, we collect lots of real-world code snippets as the ground truth and use a powerful generator to rewrite them into mutated versions. Then, we pre-train CodeEditor to edit mutated versions into the ground truth, to learn edit patterns. We conduct a large-scale study on four code editing datasets and evaluate the pre-trained CodeEditor in three settings (*i.e.,* fine-tuning, few-shot, and zero-shot). (1) In the fine-tuning setting, we train the pre-trained CodeEditor with four datasets. CodeEditor outperforms the SOTA baseline by 15%, 25.5%, 9.4%, and 26.6% on four datasets. (2) In zero-shot and few-shot settings where the fine-tuning data is limited, CodeEditor still substantially outperforms the SOTA baseline. These improvements prove that our CodeEditor is more effective in automatic code editing, and shows a strong generalization ability. In the future, we will explore more advanced pre-training techniques for automatic code editing. For example, pre-training a model to generate edit actions.

ACKNOWLEDGMENTS

This research is supported by the National Natural Science Foundation of China under Grant Nos. 62192733, 62192731, 61751210, 62072007, 61832009, and 62192730. The AI training platform supporting this work was provided by High-Flyer AI. (Hangzhou High-Flyer AI Fundamental Research Co., Ltd.)

REFERENCES

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2655–2668.

[2] Amiangshu Bosu and Jeffrey C Carver. 2013. Impact of peer code review on peer impression formation: A survey. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 133–142.

[3] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering* 48, 4 (2020), 1385–1399.

[4] Saikat Chakraborty and Baishakhi Ray. 2021. On Multi-Modal Learning of Editing Source Code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 443–455.

[5] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2019. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In *International Conference on Learning Representations*.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 4171–4186.

[7] Aryaz Eghbali and Michael Pradel. 2022. CrystalBLEU: precisely and efficiently measuring the similarity of code. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[9] GitHub. 2022. Real-world code changes. https://github.com/apache/hadoop/pull/4670/files#diff-dac9de4dd225110eff2f29a44000bf32705f02df2b3fcf17b5d89bc236c12f01.

[10] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.

[11] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019). arXiv:1909.09436 http://arxiv.org/abs/1909.09436

[12] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation,

Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 7871–7880.

[13] Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. 2021. Editsum: A retrieve-and-edit framework for source code summarization. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 155–166.

[14] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. SkCoder: A Sketch-based Approach for Automatic Code Generation. *CoRR* abs/2302.06144 (2023). https://doi.org/10.48550/arXiv.2302.06144 arXiv:2302.06144

[15] Jia Li, Zhuo Li, Huangzhao Zhang, Ge Li, Zhi Jin, Xing Hu, and Xin Xia. 2022. Poison Attack and Defense on Deep Source Code Processing Models. *CoRR* abs/2210.17029 (2022). https://doi.org/10.48550/arXiv.2210.17029 arXiv:2210.17029

[16] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2023. Towards Enhancing In-Context Learning for Code Generation. *CoRR* abs/2303.17780 (2023). https://doi.org/10.48550/arXiv.2303.17780 arXiv:2303.17780

[17] Xiaonan Li, Yeyun Gong, Yelong Shen, Xipeng Qiu, Hang Zhang, Bolun Yao, Weizhen Qi, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022. CodeRetriever: Unimodal and Bimodal Contrastive Learning. *CoRR* abs/2201.10866 (2022). arXiv:2201.10866 https://arxiv.org/abs/2201.10866

[18] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 473–485.

[19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019). arXiv:1907.11692 http://arxiv.org/abs/1907.11692

[20] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

[21] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 336–347.

[22] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. 2016. API code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 511–522.

[23] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, and Hridesh Rajan. 2013. A study of repetitiveness of code changes in software evolution. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 180–190.

[24] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-Code: Sequence-to-Sequence Pre-Training for Learning the Representation of Source Code. In *2022 IEEE/ACM 44st International Conference on Software Engineering (ICSE)*. IEEE.

[25] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[26] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[27] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21 (2020), 1–67.

[28] Baishakhi Ray, Miryung Kim, Suzette Person, and Neha Rungta. 2013. Detecting and characterizing semantic inconsistencies in ported code. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 367–377.

[29] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. 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)*. 1715–1725.

[30] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and Policy Considerations for Deep Learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 3645–3650.

[31] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8984–8991.

[32] Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. 2022. AutoTransform: Automated Code Transformation to Support Modern Code Review Process. In *2022 IEEE/ACM 44st International Conference on Software Engineering (ICSE)*. IEEE.

[33] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 25–36.

[34] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using pre-trained models to boost code review automation. In *Proceedings of the 44th International Conference on Software Engineering*. 2291–2302.

[35] Rosalia Tufano, Luca Pascarella, Michele Tufanoy, Denys Poshyvanykz, and Gabriele Bavota. 2021. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 163–174.

[36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[37] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[38] Pengcheng Yin and Graham Neubig. 2018. TRANX: A Transition-based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (Demo Track)*.

[39] Wangchunshu Zhou, Tao Ge, Canwen Xu, Ke Xu, and Furu Wei. 2021. Improving Sequence-to-Sequence Pre-training via Sequence Span Rewriting. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 571–582.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009