CoditT5: Pretraining for Source Code and Natural Language Editing

Jiyang Zhang
The University of Texas at Austin
Austin, TX, USA
jiyang.zhang@utexas.edu

Sheena Panthaplackel
The University of Texas at Austin
Austin, TX, USA
spantha@cs.utexas.edu

Pengyu Nie
The University of Texas at Austin
Austin, TX, USA
pynie@utexas.edu

Junyi Jessy Li
The University of Texas at Austin
Austin, TX, USA
jessy@austin.utexas.edu

Milos Gligoric
The University of Texas at Austin
Austin, TX, USA
gligoric@utexas.edu

ABSTRACT

Pretrained language models have been shown to be effective in many software-related generation tasks; however, they are not well-suited for editing tasks as they are not designed to reason about edits. To address this, we propose a novel pretraining objective which explicitly models edits and use it to build CoditT5, a large language model for software-related editing tasks that is pretrained on large amounts of source code and natural language comments. We fine-tune it on various downstream editing tasks, including comment updating, bug fixing, and automated code review. By outperforming standard generation-based models, we demonstrate the generalizability of our approach and its suitability for editing tasks. We also show how a standard generation model and our edit-based model can complement one another through simple reranking strategies, with which we achieve state-of-the-art performance for the three downstream editing tasks.

CCS CONCEPTS

• **Computing methodologies → Machine learning**; • **Software and its engineering → Software evolution**.

KEYWORDS

Pretrained language models, editing, bug fixing, comment updating, automated code review

ACM Reference Format:
Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2022. CoditT5: Pretraining for Source Code and Natural Language Editing. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556955

1 INTRODUCTION

Large language models pretrained on massive amounts of data have led to remarkable progress in recent years, with models like BART [26], GPT [7, 43], and T5 [44] yielding huge improvements for a vast number of text generation tasks. Inspired by this, a new research initiative has emerged around building large models that are pretrained on source code and technical text to address software-related tasks. This includes models like PLBART [1], CodeGPT-2 [32], and CodeT5 [55]. While these models demonstrate impressive performance on generation tasks like code summarization, code generation, and code translation, it is unclear if they are well-suited for the *editing* nature of many software-related tasks. For instance, bug fixing [49] entails editing source code to resolve bugs, automated code review [51] requires editing source code to incorporate feedback from review comments, and comment updating [16, 29, 31, 40] pertains to updating outdated natural language comments to reflect code changes.

In principle, such editing tasks can be framed as standard generation tasks in which an input sequence (e.g., *buggy* code snippet) is completely re-written to form the output sequence (e.g., *fixed* code snippet). In this way, existing pretrained conditional generation models can be fine-tuned to autoregressively generate a sequence from scratch. However, this can be problematic in practice [40]. When applying large generation models like PLBART and CodeT5 to these tasks, we find that they can generate output which merely copies the input without performing any edits (up to 34.25%) or even deviates substantially from the input, introducing irrelevant changes. We provide an example of automated code review in Figure 1, where a reviewer prescribes edits that need to be made to a given code snippet: "Generally better to qualify than making static import". Using the code snippet and this comment, PLBART generates an output sequence which copies the original code, without applying any edits. While the output is valid and a likely sequence according to PLBART's language model, it makes no edits based on the reviewer's comments.

We attribute these weaknesses to the fact that such models rely on pretraining objectives designed for generating code (or software-related natural language) in sequence by exploiting patterns with respect to preceding tokens. Therefore, a model has to learn to *implicitly* perform edits by generating tokens one by one in accordance with the underlying probability that it has learned for which

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556955

tokens belong alongside one another, rather than being aware of where information should be retained or modified.

Intuitively, edit-based generation requires a different approach that more frequently refers back to the input sequence, and can often be characterized by localized operations (e.g., insertion, deletion, substitution). To guide a model in discerning edit locations in the input sequence and reason about the necessary edit operations, we design a novel pretraining objective that *explicitly* models edits. Our approach is inspired by content planning in natural language generation where a skeleton of key elements are first generated and used to guide more accurate and precise generation of full text [14, 33, 42, 45]. Specifically, during decoding, a model first generates an *edit plan* that explicitly details the edit operations. Then, it proceeds to autoregressively generate the target edited sequence, during which it attends to the edit plan. Through this, we effectively encourage the model to learn to better reason about edits and how they should be applied to form the target sequence. Using this objective, we develop CoditT5, a large language model for software-related edit tasks that is pretrained on more than 5.9 million open-source programming language code snippets and 1.6 million natural language comments from the CodeSearchNet [22] training data.

For evaluation, we fine-tune CoditT5 on three downstream tasks: comment updating, bug fixing, and automated code review. For each of these tasks, we show that CoditT5 outperforms state-of-the-art models as well as large pretrained standard generation-based models. Through this, we demonstrate that our model and the proposed edit-based pretraining objective generalize across tasks and are better suited for editing tasks in the software domain.

Furthermore, in our evaluation, we find that our edit-based model, CoditT5, can be further improved if combined with a standard generation-based model. We find that the edit-based and standard generation-based models are complementary to one another. Namely, while the edit-based model provides better explicit modeling of concrete edits, a standard generation-based model provides certain advantages in terms of the contextual coherence of the generated target sequence. To exploit this complementary nature of these models, we combine the two models through reranking strategies which require no additional training. Our results show that the combined approaches outperform the two models individually by up to 19.35%.

We summarize our main contributions as follows:

- We formulate a novel pretraining objective that entails first generating a plan consisting of edit operations to be applied to the input sequence followed by the resulting target sequence.
- We build and release CoditT5, a large language model for software-related editing tasks that is pretrained on large amounts of source code and natural language with the new pretraining objective.
- Upon task-specific fine-tuning, we show that CoditT5 achieves improved performance over existing models for three distinct downstream editing tasks (comment updating, bug fixing and automated code review), demonstrating its effectiveness and generalizability.
- We show that by combining our edit-based CoditT5 model with a standard generation model through simple reranking strategies, we can beat each of the individual models and achieve new state-of-the-art in all three tasks, demonstrating the complementary nature of edit-based and standard generation models.

Before Editing

```
default List<Pattern> getExcludedResponseHeaderPatterns() {
    return emptyList();
}
```

Reviewer's Comment

Generally better to qualify than making static import

PLBART

```
default List<Pattern> getExcludedResponseHeaderPatterns() {
    return emptyList();
}
```

Figure 1: An example in automated code review task where PLBART merely copies the input which does not match reviewer's comment.

Our code and data is publicly available at https://github.com/EngineeringSoftware/CoditT5.

2 BACKGROUND

We first give a high-level overview of the building blocks that are necessary to understand our approach.

2.1 Generation with Transformer-Based Models

Conditional Sequence Generation. Conditional sequence generation entails generating an output sequence given an input sequence. Many tasks are framed in this manner, including machine translation (e.g., translating a sentence from French to English) [2], text summarization (e.g., generating a brief summary for a given news article) [46], and code generation (e.g., generating a code snippet for a given natural language specification) [58].

Encoder-Decoder Framework. In recent years, conditional sequence generation tasks are being addressed with encoder-decoder models. An encoder-decoder model consists of two neural components: an encoder and a decoder. The input sequence is fed into the encoder, which produces learned vector representations of the tokens in that sequence. These learned vector representations are then passed into the decoder, which generates the output sequence one token at a time. Specifically, the decoder predicts the next token by reasoning over the input sequence and the tokens generated at previous time steps.

Transformers. Transformers [52] are powerful neural models that are commonly adopted as the encoder and decoder in the encoder-decoder framework. These models rely on an *attention* mechanism to learn representations for tokens by relating them to other tokens in the sequence. Namely, a transformer-based encoder will learn representations for each token in the input sequence by "attending" to other input tokens. For the decoder, when generating a token at timestep t, it will "attend" to the representations of the output tokens generated from timestep 1 to $t - 1$ as well as the representations of tokens from the input sequence. Transformer models can become very large with huge numbers of attention heads, encoder and decoder layers.

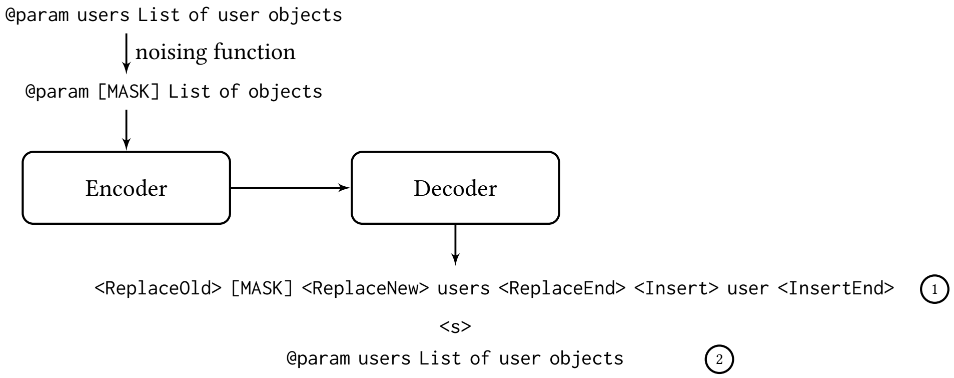
@param users List of user objects
noising function
@param [MASK] List of objects
Encoder
Decoder
<ReplaceOld> [MASK] <ReplaceNew> users <ReplaceEnd> <Insert> user <InsertEnd>
1
<s>
@param users List of user objects
2

Figure 2: The corrupted text is encoded with a bidirectional encoder, and the decoder is pretrained to generate sequences of edit actions to recover the original text followed by a separation token (<s>), and finally the target sequence

2.2 Large Pretrained Language Models

Large pretrained language models generally refer to the class of large transformer-based models that are trained on large amounts of unlabeled data (collected from webpages, news articles, etc.) with unsupervised training objectives. This includes a vast number of models like GPT [7, 43], BART [26], and T5 [44].

Denoising Autoencoder Pretraining. BART and T5 models are pretrained using denoising autoencoding unsupervised training objectives. Namely, a noising function is first applied to a given input sequence *inp* to form *inp*′. Common noising functions include *Token Masking*: tokens in the input sequence are randomly masked; *Token Deletion*: random tokens are deleted from the input sequence; *Token Infilling*: a span of tokens are sampled and replaced with a mask token; *Sentence Permutation*: sentences in the document are shuffled in a random order. Then, *inp*′ is fed into a model's encoder, and the encoder's learned representation is passed into the decoder, which generates an output sequence, *out*, that is expected to resemble the original input sequence (*inp*). In other words, the model is trained to "denoise" *inp*′, using a training objective that minimizes the error between *out* and the original input, *inp*. Through this, the model learns to extract meaning from the input sequence and also generate fluent and coherent output. Therefore, by pretraining on massive amounts of data, the model develops an understanding of how things in the world relate to one another as a strong language modeling capability.

Fine-tuning for Downstream Tasks. Since large pretrained language models are trained using unsupervised training objectives on huge amounts of data, they cannot generally be directly applied to downstream tasks (e.g., translation, summarization). Fine-tuning is a common technique to transfer the knowledge learned during pretraining to target downstream tasks. Specifically, the pretrained model is further trained for the downstream task on some amount of supervised data.

2.3 Large Pretrained Language Models for Software Engineering

Inspired by the success of large pretrained models in Natural Language Processing (NLP), a number of machine learning models pretrained on source code and technical text have been proposed for solving various software-related problems.

For instance, inspired by BART, Ahmad et al. [1] developed PLBART, which is a large pretrained language model that can be fine-tuned for a number of code understanding (e.g., code summarization) and generation (e.g., code translation) tasks. Similarly, inspired by T5, Wang et al. [55] built a larger model CodeT5, which is pretrained on six programming languages together with their natural language comments collected from open-source repositories. Specially, it is pretrained to incorporate information from identifiers in the code. CodeT5 has shown promising results in code-related generation tasks such as code summarization, code generation and code-related understanding tasks such as clone detection and vulnerability identification. However, aforementioned models are for generation and they are only implicitly aware of edit operations if at all.

3 CODITT5

CoditT5 is built upon the encoder-decoder framework with the same architecture as CodeT5. As shown in Figure 2, the model is pretrained with our proposed objective: generating the edit-based output sequence given the corrupted input sequence. In this section, we first explain our proposed pretraining objective (Section 3.1). We then discuss how we build CoditT5 by pretraining on this objective, including the data used for pretraining (Section 3.2), and additional details of the pretraining setup (Section 3.3).

3.1 Pretraining Objective

We formulate a new pretraining objective that is designed to encourage a model to explicitly reason about edits. At a high-level, this objective falls under the realm of denoising autoencoding in which an input sequence is first corrupted with noising functions and the model is trained to *denoise* the corrupted sequence by generating an output sequence that matches the original input sequence. While existing models like PLBART and CodeT5 pretrained using this setup perform very well on various generation tasks (e.g., code summarization/generation), we find that they do not generalize well when fine-tuned on editing tasks. Namely, they are susceptible to learning to copy the original input sequence instead of actually performing edits, up to 34.25% of the time (Table 3).

We propose the following *edit-based output sequence* representation (shown in Figure 2): [Edit Plan] <s> [Target Sequence], where the model is trained to generate an *edit plan* (①) consisting of explicit edit operations that must be applied to the corrupted sequence to reconstruct the original input sequence, followed by a separation token (<s>), and finally the *target sequence* (②) that matches the original input sequence. This is inspired by the concept of *content planning*, originating from natural language generation [45]. In content planning, a high-level plan is first outlined, specifying the discourse structure of the content to be generated, and then lexical realization is performed to generate the text.

3.1.1 Edit Plan. The edit plan entails the specific edit operations that are needed to recover the original input sequence. For example, in Figure 2, the input sequence: "`@param` users List of user objects" is corrupted by masking "users" and removing token "user": "`@param` `[MASK]` List of objects". With this, a model must first reason about the fact that `[MASK]` in the corrupted input sequence needs to be replaced with "users" and "user" should be inserted between "of" and "objects" when producing the target sequence. To construct the sequence of edit operations, we closely follow the format proposed by Panthaplackel et al. [40]:

```
<Operation> [span of tokens] <OperationEnd>
```

Here, `<Operation>` is either `Insert` or `Delete`. We also include the `Replace` operation, with a slightly different structure (since both the old content to be replaced as well as the new content to replace it with must be specified):

```
<ReplaceOld> [span of old tokens]
<ReplaceNew> [span of new tokens] <ReplaceEnd>
```

To determine the specific edit operations for a given example, we use difflib[1] to compute the optimal set of edits needed to transform the corrupted input sequence into the original input sequence. Multiple edit operations are placed in the same order as the span of tokens under editing appears in the input sequence (for example, the edit plan in Figure 2 consists of two edit operations).

3.1.2 Target Sequence. One might ask whether we could simply apply the sequence of edit operations in the generated edit plan to the corrupted input sequence directly to recover the original input sequence *heuristically*. For example, if we align "`<ReplaceOld> [MASK] <ReplaceNew> user <ReplaceOld>`" with a corrupted input sequence "`@param` `[MASK]` List of user objects", it is very clear that all we need to do is replace `[MASK]` with "user"and no additional generation is needed. However, there are two main issues with this. First, not all operations will be specified in a deterministic manner. For example, if the edit plan is "`<Insert> user <InsertEnd>`", it is not clear where the new token "user" should be added to. Second, the generated edit plan does not correspond to contiguous output tokens since it consists of fragmented information (edit operations and token spans) rather than a complete sentence. As a result, neural language models may fail to generate correct edit plans due to their lack of language properties such as fluency and coherency [40].

Table 1: Statistics collected from downstream tasks for creating pretraining dataset. Avg. No. of Tokens represents the average number of tokens in each edited span; Avg. No. of Spans represents the average number of edited spans in each input sequence.

	PL	NL
Probability of `Delete` edit	0.49	0.07
Probability of `Insert` edit	0.21	0.11
Probability of `Replace` edit	0.30	0.82
Avg. No. of Tokens	6.50	3.00
Avg. No. of Spans	1.90	1.40

Therefore, we need an additional step for *learning* to apply edits while simultaneously maintaining fluency and coherency. For this reason, once the edit plan is outlined as a sequence of edit operations, the target sequence (which is expected to recover the original input sequence) must also be generated: "`@param` users List of user objects". The decoder generates tokens in a left-to-right manner, meaning that when generating a token at a given timestep, it is aware of all tokens generated in previous timesteps. So, when generating the target sequence, the decoder can exploit the sequence of edits that was generated in the edit plan earlier. In this way, the model can reason the edits and the generation simultaneously.

3.1.3 Noising Functions. To support learning across a diverse set of edit actions during pretraining, we consider multiple noising functions for corrupting the input sequence: 1) randomly masking spans with the special `[MASK]` token which requires the model to replace it with the correct spans, 2) inserting `[MASK]` token at random positions which requires the model to identify the useless spans and delete them and 3) deleting spans of tokens in the input sequence which requires the model pinpoint the position and add back the missing pieces.

3.2 Pretraining Data

3.2.1 Data Collection. Following prior work, we pretrain CoditT5 on large amounts of source code and natural language comments from the CodeSearchNet [22] dataset which consists of functions of six programming languages (Java, Python, Ruby, Php, Go and JavaScript) together with the natural language comments. CodeSearchNet is widely used to pretrain large language models, such as CodeT5 [55] and UniXcoder [18]. We use the training set of the processed CodeSearchNet dataset provided by Guo et al. [18] which contains 6.1 million programming languages code snippets (functions/methods) and 1.9 million natural language comments.

3.2.2 Data Preparation. To enable CoditT5 to capture common edit patterns, we want the pretraining dataset to reflect the common activities conducted by software developers. Specifically, in the pretraining dataset, the probability of each edit operations applied to the spans in the input sequence and the length (number of tokens) of the corrupted span should be consistent with the distributions and sizes of real-world edits in downstream editing tasks.

[1] https://docs.python.org/3/library/difflib.html

Table 2: Statistics of the datasets used to pretrain CoditT5. First row: number of programming language and natural language; second row: average number of tokens in corrupted input sequences; third row: average number of tokens in the output sequence (edit plan + target sequence).

	PL	NL
Examples	5,956,069	1,675,277
Avg. C-Tokens	102.01	15.42
Avg. O-Tokens	120.23	26.57

To this end, we collect statistics for source code edits from the training sets of the bug fixing and automated code review downstream tasks and statistics for natural language edits from the comment updating's training set. As shown in Table 1, we collect the probability of each edit operation (insert, delete and replace) to be performed on a span; the average number of tokens in each span that is edited; and the average number of spans that are edited in each input sequence. For each example in the pretraining dataset, we then uniformly sample the spans and the edit operations that should be applied in accordance with the statistics collected from the downstream datasets.

Similar to CodeT5 [55], we use the RoBERTa [30] tokenizer to tokenize all sequences (input, edit plan, target). More concretely, the tokenizer splits words in the sequence into *tokens* (subwords) that are used by the model. Moreover, we remove input sequences that are shorter than 3 tokens and longer than 512 tokens after tokenization which leave us with 5.9 million programming language code snippets and 1.6 million natural language comments. This is because too short inputs are usually incomplete and CodeT5 is designed to only handle sequence of length 512. Table 2 presents the statistics of the pretraining dataset.

3.3 Pretraining Setup

Model Architecture. CoditT5 consists of 12 encoder and decoder layers, 12 attention heads, and a hidden dimension size of 768. The total number of parameters is 223M. Model parameters are initialized from the CodeT5-base model, and we further pretrain it on the CodeSearchNet pretraining dataset (Section 3.2) using our proposed objective (Section 3.1).

Training. We implement CoditT5 using PyTorch 1.9.0 and use 16 NVidia 1080-TI GPUs, Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz for pretraining for 4 days. For fine-tuning, we run the experiments on 4 NVidia 1080-TI GPUs, Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz with the same hyper-parameters as CodeT5.

4 EXPERIMENTAL DESIGN

To assess CoditT5 and our proposed pretraining objective, we fine-tune the model on three software-related downstream tasks. Note that during fine-tuning, the model is still trained to generate the edit-based output sequence. However, at test time, we discard the edit plan and take the generated target sequence as the final model output. Namely, we use the generated sequence after the separation token <s> as model's prediction.

Table 3: Percentage that model just copy the input.

Models	PLBART	CodeT5	CoditT5
$B2F_s$	6.48	7.97	0.55
$B2F_m$	10.92	10.08	0.78
Comment Updating (clean)	21.33	16.67	2.67
Comment Updating (full)	34.25	25.47	5.73
Automated Code Review	22.24	29.28	1.28

Table 4: Statistics for the datasets used for downstream tasks.

Task		Train	Valid	Test
Comment Updating	clean	16,494	1,878	150
	full	16,494	1,878	1,971
Bug Fixing	$B2F_s$	46,628	5,828	5,831
	$B2F_m$	52,324	6,542	6,538
Automated Code Review		13,753	1,719	1,718

4.1 Downstream Tasks

Comment Updating. The task of comment updating entails automatically updating a natural language comment to reflect changes in the corresponding body of code [40]. For instance, in Example 2 in Figure 5, the old `@return` comment needs to be revised based on the changes in the method. Instead of directly returning the yaw Euler angle measured in radians, the unit of the return value is changed to degrees in the new version, with the method call `Math.toDegrees()`.

Bug Fixing. Given a *buggy* code snippet, the task of bug fixing entails generating a *fixed* code snippet, which no longer contains the bug [48].

Automated Code Review. Given a code snippet under review and a brief natural language sentence prescribing code edits, automated code review requires automatically generating the revised code snippet, which captures the recommended changes [51]. For example, in Figure 1, `emptyList()` should be changed to `Collections.emptyList()` because the reviewer suggests *not* using static import.

4.2 Data for Downstream Tasks

We use datasets that have been established and previously used for each of the three tasks. The statistics of the datasets is shown in Table 4. Unlike pretraining where the goal is to recover the corrupted input sequences, during fine-tuning, CoditT5 is trained to generate an edit plan for completing the downstream editing task, that can be applied to a part of the input (e.g., old comment), followed by the target sequence (e.g., new comment).

Comment Updating. For this task, Panthaplackel et al. [39] has released a corpus of Java method changes paired with changes in the corresponding comments (spanning `@return`, `@param`, and summary comments). This dataset also comes with a *clean* subset of the test set which was manually curated. The input sequence used for fine-tuning is formed by concatenating the old comment and code edits. The code edits follow the representation described in Section 3.1.1, except that an additional Keep operation is included to denote spans that are left unchanged.

Bug Fixing. We consider the Java BugFixPairs-Small ($B2F_s$) and BugFixPairs-Medium ($B2F_m$) datasets, originally released by Tufano et al. [48]. Chakraborty and Ray [8] supplemented these datasets with additional context, namely natural language guidance from the developer, and the method where the patch should be applied. $B2F_s$ contains shorter methods with a maximum token length 50, and $B2F_m$ contains longer methods with up to 100 tokens in length. The input sequence used for fine-tuning is formed with the buggy code, natural language guidance, and code context.

Automated Code Review. We use the automated code review dataset released by Tufano et al. [51], which consists of Java methods (before and after the review) paired with pull request comments, derived from pull request reviews on GitHub and Gerrit. To reduce the vocabulary size, they further abstracted Java methods by replacing identifiers and literals with special tokens. In this work, we use the data with concrete tokens. The input sequence used for fine-tuning is formed using the code snippet before review and the pull request comment from reviewers.

4.3 Baselines

4.3.1 Generation Baselines. We consider two large standard generation language models trained with denoising autoencoding pretraining objectives which are not edit-based: **PLBART** and **CodeT5**. Both of these are fine-tuned to directly generate the target output sequence. Furthermore, to better assess the value of actually pretraining using the proposed objective instead of simply fine-tuning a model to generate an edit-based output sequence, we also consider fine-tuning CodeT5 to generate the specialized edit-based output sequence representation. We refer to this as **CodeT5 (w/ edit-based output)**. We fine-tune each of these models using the same input context as CoditT5.

4.3.2 Task-Specific Baselines. We additionally compare against the state-of-the-art models for each of the downstream tasks.

For comment updating, the state-of-the-art model is **Panthaplackel et al. [40]**, which entails Recurrent Neural Network (RNN) based encoders for representing the old comment and code edits, and an RNN-based decoder for decoding edits. These edits are parsed at test time and reranked based on similarity to the old comment and likelihood based on a comment generation model.

For bug fixing, the state-of-the-art model is essentially PLBART fine-tuned on the $B2F_s$ and $B2F_m$ to generate the fixed code [8].

For automated code review, no baselines are available for the specific version of the dataset we used with concrete identifiers and literals (rather than the one with abstracted identifiers and literals). Therefore, we rely on those described in Section 4.3.1 and establish new baselines for this version of the dataset.

4.4 Evaluation Metrics

For comment updating, we report performance on the same metrics that have been used previously to benchmark models for this task [40]. This includes: xMatch (whether the model prediction *exactly matches* the ground truth), common metrics that measure lexical overlap for evaluating text generation (BLEU-4 [2] [41] and METEOR [3]), and common metrics for measuring text editing (GLEU [35] and SARI [56]). For bug fixing, we use xMatch, as done in prior work [8]. For automated code review, we report performance on xMatch and BLEU-4, which have been used previously to benchmark models for this task [51].

Before Editing:

```
public HashConfigurationBuilder capacityFactor (float capacityFactor) {
    if ( numSegments < 0 )
        throw new IllegalArgumentException ("capacityFactor must be positive");
    this.capacityFactor = capacityFactor ;
    return this;
}
```

Reviewer's Comment:

typo: capacityFactor instead of numSegments

CodeT5:

```
public HashConfigurationBuilder capacityFactor(float capacityFactor) {
    this.capacityFactor = capacityFactor;
    return this;
}
```

CoditT5:

```
public HashConfigurationBuilder capacityFactor (float capacityFactor) {
    if ( capacityFactor < 0 )
        throw new IllegalArgumentException ("capacityFactor must be positive") ;
    this.capacityFactor = capacityFactor;
    return this;
}
```

Figure 3: Comparing the output of CodeT5 and CoditT5 for a automated code review example. CodeT5 generates incorrect output that drastically deviates from the input code while CoditT5 generates the correct output, performing only relevant edits.

5 EVALUATION

We organize our evaluation around three main research questions:
RQ1: How does our edit-based model, CoditT5, compare to generation and task-specific baselines for edit-related tasks?
RQ2: Does our proposed pretraining objective help a model in better reasoning about and performing edits?
RQ3: Can a standard generation model complement CoditT5 by integrating the two models?

5.1 Comparing CoditT5 to Baselines

We present results in Tables 5-8. Note that the results shown in the last two rows in each of the tables are explained later in Section 5.3. We perform statistical significance testing using bootstrap tests [4] with confidence level 95%.

> **RQ1:** How does our edit-based model, CoditT5, compare to generation and task-specific baselines for edit-related tasks?

We find that CoditT5 (and most of the pretrained models) drastically outperforms Panthaplackel et al. [40] (a non-pretrained model) across metrics for comment updating. This demonstrates the value of large language model pretrained on vast amounts of data using unsupervised pretraining objectives.

[2] We measure 1~4-gram overlap and compute the average.

Table 5: Results for comment updating on the clean test set. The results with the same prefixes (e.g., β) are NOT statistically significantly different.

Models	xMatch	BLEU-4	METEOR	GLEU	SARI
Panthaplackel et al. [40]	33.33	56.55	52.26	51.88	56.23
PLBART	35.33	62.04^{γ}	56.79	54.75	52.83
CodeT5	38.00	65.20^{α}	59.63	58.84^{β}	58.80
CodeT5 (w/ edit-based output)	40.00	62.97^{γ}	59.08	58.72^{β}	$61.11^{\epsilon\eta}$
CoditT5	43.33^{χ}	64.56	60.75	59.53	$61.41^{\delta\epsilon}$
CoditT5 (reranked with CodeT5)	**45.33**	**66.80**	**63.33**	**61.60**	$61.48^{\delta\eta}$
CodeT5 (reranked with CoditT5)	44.00^{χ}	65.58^{α}	62.44	60.48	**62.57**

Table 6: Results for comment updating on the full test set. The results with the same prefixes (e.g., β) are NOT statistically significantly different.

Models	xMatch	BLEU-4	METEOR	GLEU	SARI
Panthaplackel et al. [40]	24.81	48.89	44.58	45.69	47.93
PLBART	22.98	$55.42^{\hbar\iota}$	49.12	47.83	43.40
CodeT5	28.56	58.37^{α}	53.13	51.90	49.23
CodeT5 (w/ edit-based output)	$29.83^{\delta\gamma}$	54.83	50.71	50.67^{ϵ}	52.01^{η}
CoditT5	29.38^{δ}	$55.30^{\beta\iota}$	51.14^{χ}	50.62^{ϵ}	51.39
CoditT5 (reranked with CodeT5)	$\mathbf{30.14}^{\gamma}$	$\mathbf{58.72}^{\alpha}$	**53.60**	**52.81**	50.47
CodeT5 (reranked with CoditT5)	27.80	$55.54^{\beta\hbar}$	51.44^{χ}	50.02	$\mathbf{52.24}^{\eta}$

Table 7: Results on bug fixing dataset. The results with the same prefixes (e.g., β) are NOT statistically significantly different.

Models	xMatch	
	$B2F_s$	$B2F_m$
PLBART	31.09	24.18
CodeT5	34.81	26.66
CodeT5 (w/ edit-based output)	36.37	29.28^{α}
CoditT5	37.52	29.96^{α}
CoditT5 (reranked with CodeT5)	**40.22**	$\mathbf{32.06}^{\beta}$
CodeT5 (reranked with CoditT5)	39.56	32.24^{β}

Table 8: Results for automated code review. The results with the same prefixes (e.g., β) are NOT statistically significantly different.

Models	xMatch	BLEU-4
PLBART	26.78	79.38
CodeT5	34.98	83.20
CodeT5 (w/ edit-based output)	36.38^{α}	80.06^{β}
CoditT5	37.19^{α}	80.50^{β}
CoditT5 (reranked with CodeT5)	40.98	$\mathbf{84.12}^{\chi}$
CodeT5 (reranked with CoditT5)	**43.42**	83.92^{χ}

Next, across all three tasks, CoditT5 achieves higher performance than the two standard generation-based pretrained models, significantly outperforming PLBART and CodeT5 for most of the metrics, highlighting the benefit of explicitly modeling edits for these editing tasks. In fact, CodeT5 (w/ edit-based output), which explicitly models edits only during fine-tuning rather than pretraining, outperforms CodeT5 on edit-based metrics (xMatch, SARI). This further underlines the utility of the edit-based output sequence representation that we developed.

Nonetheless, across most metrics, CoditT5 still outperforms CodeT5 (w/ edit-based output), which is not pretrained using the pretraining objective but uses the same edit-based output sequence representation during fine-tuning. This demonstrates the importance of actually pretraining with this representation rather than relying on fine-tuning alone.

5.2 Evaluating our Pretraining Objective

While we observe that CoditT5 tends to achieve slightly lower performance than CodeT5 on generation-based metrics (BLEU-4, METEOR) for two of the tasks, we find that it significantly outperforms other metrics which capture whether the correct edits are generated, such as xMatch and GLEU and SARI for comment updating. This suggests that CoditT5 is indeed better at *editing*. By inspecting the outputs of the two models, we find that CodeT5 tends to make drastic and unnecessary edits while CoditT5 appears to be better at making more fine-grained edits. For example, in Figure 3,

Example 1

Before Editing:

```
protected boolean isProcessed(ChronicleLogOffsetTracker tracker, long offset) {
    long last = tracker.readLastCommittedOffset();
    return ( last > 0 ) && ( last >= offset );
}
```

Reviewer's Comment:

No need for parentheses.

Edit plan

```
⟨Delete⟩ ( ⟨Delete_End⟩ ⟨Delete⟩ ) ⟨Delete_End⟩
```

Target sequence:

```
protected boolean isProcessed(ChronicleLogOffsetTracker tracker, long offset) {
    long last = tracker.readLastCommittedOffset();
    return last > 0 && last >= offset;
}
```

Example 2

Before Editing:

```
public Builder setDataSize(Estimate dataSize) {
    this.dataSize = requireNonNull(dataSize, "dataSize can not be null");
    return this;
}
```

Reviewer's Comment

you don't validate in other builders method (and you don't have to)

Edit plan

```
⟨Delete⟩ equireNonNull(dataSize, "dataSize can not be null"); ⟨Delete_End⟩
```

Target sequence:

```
public Builder setDataSize(Estimate dataSize) {
    this.dataSize = dataSize;
    return this;
}
```

Figure 4: Examples for automated code review for which CoditT5 generated ambiguous or erroneous edit plans but still managed to generate the correct target sequences.

Table 9: Percentages of target sequence generated by CoditT5 being consistent with the edit plan.

Datasets	Is Consistent (%)
$B2F_s$	92%
$B2F_m$	88%
Comment Updating (clean)	87%
Comment Updating (full)	85%
Automated Code Review	74%

CodeT5 generates output that completely discards critical statements in the code, whereas CoditT5 is able to correctly localize the part of the input code that needs to be changed and make editions properly. We attribute this to the fact that CodeT5 is not designed to reason about edits while CoditT5 is. We further evaluate the influence of our proposed pretraining objective on this editing capability.

RQ2: Does our proposed pretraining objective help a model in better reasoning about and performing edits?

First, we compare how often CoditT5 naively copies the input content without actually performing any edits, to two pretrained models which use generation-based pretraining objectives. We report the percentages in Table 3. By copying substantially less often than the PLBART and CodeT5, we find that CoditT5 learns to more frequently perform edits with our proposed edit-based pretraining objective which indicates it is suitable for editing tasks.

CoditT5's decoder is encouraged to generate a target sequence that follows the outlined edit plan; however, we do not constrain the decoder in any way to do this.[3] Nonetheless, we find that in the majority of cases (74%-92%), the target sequence is consistent with the edit plan, as shown in Table 9. More concretely, the target sequence generally resembles what would be produced if the edit operations in the edit plan were applied to the original content. This suggests that the pretraining objective does in fact guide the model in reasoning about edits.

For cases in which there is ambiguity or errors in the edit plan, we find that CoditT5 still often manages to generate the correct target sequence, by disregarding unreasonable edits or disambiguating ambiguous edits. We show two examples in automated code review in Figure 4 with the Java method before review, the generated edit plan, and the generated target sequence. In Example 1, the edit plan is ambiguous since there are multiple instances of "(" and it does not specify which one(s) should be deleted. However, the generated target sequence is correct, as the model was able to correctly reason about the most appropriate edit locations. In Example 2, the edit plan is imprecise and blindly following this plan would result in syntactically incorrect code, but the model still managed to perform the correct edits and produced valid output by ignoring the fallacious edit. Overall, we find that both components of the edit-based output sequence representation used in the pretraining objective (edit plan and target sequence) are critical.

5.3 Integrating CoditT5 and CodeT5

CoditT5 is designed to complement a generation model by providing more explicit guidance for edits. However, a model that is trained to generate edits can struggle with coherence and fluency since it is not actually trained to generate consecutive text [40]. By including the generation of the target sequence in the pretraining objective, we do mitigate this to some extent, even when there are ambiguities or errors in the edit plan. However, there appears to be a trade-off between performing the correct edits while maintaining performance with respect to generation metrics. More specifically, in Tables 5-8, CoditT5 outperforms CodeT5 with respect to xMatch (and SARI for comment updating), but underperforms with respect to BLEU-4. To exploit the slight superiority of CodeT5 in this respect, we consider incorporating CodeT5 into our approach.

RQ3: Can a pure generation model complement CoditT5 by integrating the two models?

5.3.1 Experimental Setup. We combine the two models using simple likelihood-based reranking strategies at test time (with no additional training). Namely, at test time, CoditT5 and CodeT5 each generate 20 candidates using beam search. While we have been only looking at the top one prediction for all previous experiments, we will consider all 20 candidates for reranking. We compute a

[3] We do not want potential errors in the edit plan to propagate to the target sequence.

Example 1

Buggy Code

```
public List<TagVFilter> getFilters() {
    if ((filters) == null ) {
        filters = new ArrayList<TagVFilter>();
    }
    return filters;
}
```

CoditT5:

```
public List<TagVFilter> getFilters() {
    if ((filters) == null ) {
        filters = new ArrayList<TagVFilter>();
    }
    return new ArrayList(filters);
}
```

CoditT5 (reranked with CodeT5):

```
public List<TagVFilter> getFilters() {
    if ((filters) == null ) {
        filters = new ArrayList<TagVFilter>();
    }
    return new ArrayList<TagVFilter>(filters);
}
```

Example 2

```
/** @return double The yaw Euler angle. */
public double getRotY() {
    return mOrientation.getRotationY();
}

/** @return ? */
public double getRotY() {
    return Math.toDegrees(mOrientation.getRotationY());
}
```

CodeT5: @return double The yaw Euler angle.

Reranked CodeT5: @return double The yaw Euler angle in degrees.

Figure 5: Examples from comment updating and bug fixing which demonstrate the impact of reranking.

reranking score for each of these to essentially re-score them. The candidate which has the highest reranking score will be the final model prediction. We investigate two different reranking strategies:

CoditT5 (reranked with CodeT5): To exploit the language-specific norms learned by CodeT5, we rerank the candidates generated by CoditT5 based on the probability score CodeT5's language model assigns to the corresponding target sequences (namely after <s>).

We compute the length-normalized conditional log probability score of CodeT5 generating the target sequence, conditioned on the same input:

$$score = log(P(T|I)^{\frac{1}{N}})$$

where T is the target sequence, I is the model's input, N is the length of T. We also length-normalize the log probability of the candidate, as scored by CoditT5, and then add the two probability scores together to obtain the reranking score.

CodeT5 (reranked with CoditT5): Conversely, we also rerank the output of CodeT5 based on the likelihood of CoditT5, such that the generated sequence can be assessed in terms of explicit edits. We first parse the output of CodeT5 into the edit-based output sequence representation (as described in Section 3.1.1) and then concatenate it with the model's output using <s>. Then we compute the likelihood of CoditT5 generating this sequence, conditioned on the same input. We then add the length-normalized log probability score of CoditT5 with the score originally assigned by CodeT5 (after length-normalizing and applying log).

5.3.2 Results. We provide results in the bottom two rows of Tables 5-8. By reranking the output of CoditT5 using CodeT5, we are able to achieve improved performance on all the metrics including BLEU-4 across tasks (and the other generation-based metric, METEOR, for comment updating). To illustrate this, consider Example 1 in Figure 5, with a buggy code snippet and outputs corresponding to CoditT5 before and after reranking. We observe that CoditT5 correctly localizes the bug and correctly identifies that the edit entails initializing an `ArrayList` in the return statement. However, the generated target sequence is a defective code snippet which does not properly initialize an `ArrayList` with the correct type `TagVFilter`. By leveraging CodeT5's likelihood score, we are able to effectively filter out the defective prediction and obtain the correct output.

By reranking the output of CodeT5 using CoditT5, we see significant improvements with respect to CodeT5 on metrics that more directly evaluate whether the correct edits were performed, including xMatch as well as GLEU and SARI for comment updating. This suggests that the edit-based and generation-based models are indeed complementary to one another. As a case study, consider Example 2 in Figure 5. CodeT5 produces a sequence which simply copies the old comment, without capturing the code changes. While this may be a likely comment sequence, according to CodeT5's language model, copying without applying any edits is not a likely edit plan to be generated for CoditT5.

By combining CoditT5 and CodeT5 through reranking, we can further boost performance substantially across most metrics for all three tasks, outperforming the two models individually, and achieving new state-of-the-art.

6 LIMITATIONS

Other Programming Languages. The downstream editing tasks we studied in this work are using Java. Since CoditT5's pretraining is on the dataset consisting of six programming languages, we expect it to also perform well on editing tasks in other programming languages, but we leave empirically verifying this as future work.

Data Contamination. CoditT5 is pretrained on data collected from open-source projects. It is possible that similar examples in pretraining data exist in downstream tasks' test set. While prior work [7] has shown that data contamination may have little impact on the performance of pretrained models in natural language processing tasks, future work can investigate this problem for pretrained models for software engineering.

7 RELATED WORK

In this section, we consider the most closely related work on learning edits, large pretrained models for code, pretrained models for code edits and combining complementary models.

Learning Edits. Prior work has studied learning edits in both natural language and programming language. We followed the approach of explicitly representing edits as sequences with edit actions. Our edit representation is inspired by Panthaplackel et al. [39, 40], who studied learning comment edits based on code edits. Brody et al. [6], Chen et al. [10], Tarlow et al. [47], Yao et al. [57] represented code as ASTs (abstract syntax trees) and the code edits as edit actions over the AST nodes rather than tokens. We do not focus on editing structured data (AST) as it can not be generalized to natural language, and it can not be easily combined with large pretrained models which are primarily based on sequence of tokens.

Alternatively, edits can be encoded into vector representations (or embeddings). Guu et al. [20] studied learning edit embeddings for natural language generation in a prototype-then-edit style. Yin et al. [59] studied learning code edits as embeddings and then applying them to natural language insertion and code bug fixing. Hashimoto et al. [21] developed a retrieve-and-edit framework for text-to-code generation, where the edits are learned as parameters of a seq2seq model. Similarly, Li et al. [27] proposed a retrieve-and-edit framework for code summarization task where the model first learns an edit vector and then generate the revised summary conditioned on it. Although learning edits as embeddings can be effective for individual tasks, it is not suitable to be used in the pretraining fine-tuning paradigm, because there is a large domain gap between the edit embeddings learned on different tasks. More over, edit embeddings are less explainable compared to the explicit edit representations we use.

Another line of work that carries out the idea of learning edits is copying mechanism, including copying individual tokens [17, 53] and spans [38, 60], which helps the model to "keep" unchanged tokens and focus on generating the edited part. Iv et al. [23] built a T5-based model to update the existing articles based on the given new evidence. The model is trained to output a *copy* token instead of the copied sentence and a special *reference* token before the updated text which identifies the evidence to support the update. Ding et al. [12] trained the model to emit pointers that indicate the positions for editions and new tokens to be inserted at the same time. Similarly, Chen et al. [10], Tarlow et al. [47] augmented the transformer-based decoder with pointers to the input graph representation of the code which specify the input locations to edit. Although related, it is orthogonal to our work of learning edits with pretraining.

Large Pretrained Models for Code. Motivated by the success of large pretrained models for many NLP tasks, domain-specific models that are pretrained on source code and technical text have emerged, including CodeBERT [15], GraphCodeBERT [19], CodeGPT-2 [32], CodeT5 [55], PLBART [1], PyMT5 [11], SynCoBERT [54], SPT-Code [37], Codex [9] and UniXcoder [18]. CodeBERT, GraphCodeBERT and SynCoBERT consists of a pretrained encoder which is designed to learn code representation. CodeGPT-2 and Codex consists of a GPT-styple [43] language model pretrained on public available code to support code completion and code generation tasks. CodeT5, PLBART, PyMT5 and SPT-Code are based on the encoder-decoder architecture for both code understanding and generation.

Similar to our approach, GraphCodeBERT, CodeT5, SynCoBERT, SPT-Code and UniXcoder also designed specialized pretraining objectives driven by their targeted tasks. As we showed in this work, the combination of an edit-based language model and a standard language model can achieve better performance than using the standard language model alone.

Pretrained Models for Code Edits. Prior work already explored applying pretrained models, despite not well-suited, on editing tasks. Chakraborty and Ray [8] used PLBART for code bug fixing, which we compared to in our work. Similarly, Drain et al. [13] further pretrained BART model on 67K Java repositories mined from GitHub and fine-tuned specifically on the bug fixing dataset [49]. Mastropaolo et al. [34], Wang et al. [55] both pretrained T5 model on CodeSerchNet and used it for bug fixing, which we included as a baseline (CodeT5). Codex [9] showed promising performance on editing tasks by specifying the existing code as a prompt and providing an edit instruction to the model. Tufano et al. [50] and Li et al. [28] both proposed a transformer-based encoder-decoder model pretrained on large code reviewer specific data for code review related tasks including code change quality estimation, review comment generation and code refinement. While they demonstrate impressive performance on various tasks, none of them are fundamentally well-suited for edit tasks. In this work, we develop CoditT5 with a novel pretraining objective for generating edit sequences, which can complement the generation model such as CodeT5 for edit tasks.

Combining Complementary Models. We used reranking [24, 36] to combine complementary models in this work. Ensembling [25] is another approach for combining complementary models for generation tasks, but requires additional training. Co-training [5] and tri-training [61] approaches, although shown to be very effective in combining complementary models, are designed for classification models rather than generation models.

8 CONCLUSION

In this paper, we present a novel edit-driven pretraining objective and use it to develop CoditT5, a pretrained language model for software-related editing tasks. CoditT5 is pretrained on large amounts of source code and natural language comments to perform edits, and we evaluate this model by fine-tuning it on three distinct downstream tasks: comment updating, bug fixing and automated code review. By outperforming task-specific baselines and pure generation baselines across tasks, we demonstrate the suitability of CoditT5 (and our pretraining objective) for editing tasks and its generalizability. We additionally find that a pure generation-based model and CoditT5 can complement one another through simple reranking strategies, which outperform each of the models individually and also achieve new state-of-the-art performance for the three downstream editing tasks that we consider.

ACKNOWLEDGMENTS

We thank Nader Al Awar, Yu Liu, Raymond J. Mooney, Aditya Thimmaiah, Zhiqiang Zang, and the anonymous reviewers for their comments and feedback. We acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. This work is partially supported by the US National Science Foundation under Grant Nos. CCF-1652517, CCF-2107291, IIS-2107524 and IIS-2145479.

REFERENCES

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

[2] Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*.

[3] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. 65–72.

[4] Taylor Berg-Kirkpatrick, David Burkett, and Dan Klein. 2012. An Empirical Investigation of Statistical Significance in NLP. In *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. 995–1005.

[5] Avrim Blum and Tom Mitchell. 1998. Combining labeled and unlabeled data with co-training. In *Computational Learning Theory*. 92–100.

[6] Shaked Brody, Uri Alon, and Eran Yahav. 2020. A structural model for contextual code changes. *International Conference on Object-Oriented Programming, Systems, Languages, and Applications* 4 (2020), 1–28.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*. 1877–1901.

[8] Saikat Chakraborty and Baishakhi Ray. 2021. On Multi-Modal Learning of Editing Source Code. In *Automated Software Engineering*. 443–455.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[10] Zimin Chen, Vincent J Hellendoorn, Pascal Lamblin, Petros Maniatis, Pierre-Antoine Manzagol, Daniel Tarlow, and Subhodeep Moitra. 2021. PLUR: A unifying, graph-based view of program learning, understanding, and repair. In *Advances in Neural Information Processing Systems*. 23089–23101.

[11] Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: multi-mode translation of natural language and Python code with transformers. In *Empirical Methods in Natural Language Processing*. 9052–9065.

[12] Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J Hellendoorn. 2020. Patching as Translation: the Data and the Metaphor. In *Automated Software Engineering*. 275–286.

[13] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. 2021. Generating bug-fixes using pretrained transformers. In *International Symposium on Machine Programming*. 1–8.

[14] Angela Fan, Mike Lewis, and Yann Dauphin. 2019. Strategies for Structuring Story Generation. In *Annual Meeting of the Association for Computational Linguistics*. 2650–2660.

[15] 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 *Empirical Methods in Natural Language Processing: Findings*. 1536–1547.

[16] Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Thomas Zimmermann. 2021. Automating the removal of obsolete TODO comments. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 218–229.

[17] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor OK Li. 2016. Incorporating Copying Mechanism in Sequence-to-Sequence Learning. In *Annual Meeting of the Association for Computational Linguistics*. 1631–1640.

[18] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Annual Meeting of the Association for Computational Linguistics*. 7212–7225.

[19] 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*.

[20] Kelvin Guu, Tatsunori B Hashimoto, Yonatan Oren, and Percy Liang. 2018. Generating sentences by editing prototypes. *Transactions of the Association for Computational Linguistics* 6 (2018), 437–450.

[21] Tatsunori B Hashimoto, Kelvin Guu, Yonatan Oren, and Percy S Liang. 2018. A retrieve-and-edit framework for predicting structured outputs. In *Advances in Neural Information Processing Systems*.

[22] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[23] Robert Iv, Alexandre Passos, Sameer Singh, and Ming-Wei Chang. 2022. FRUIT: Faithfully Reflecting Updated Information in Text. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 3670–3686.

[24] Reno Kriz, João Sedoc, Marianna Apidianaki, Carolina Zheng, Gaurav Kumar, Eleni Miltsakaki, and Chris Callison-Burch. 2019. Complexity-Weighted Loss and Diverse Reranking for Sentence Simplification. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 3137–3147.

[25] Alexander LeClair, Aakash Bansal, and Collin McMillan. 2021. Ensemble Models for Neural Source Code Summarization of Subroutines. In *International Conference on Software Maintenance and Evolution*. 286–297.

[26] 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 *Annual Meeting of the Association for Computational Linguistics*. 7871–7880.

[27] 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 *Automated Software Engineering*. 155–166.

[28] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. 2022. CodeReviewer: Pre-Training for Automating Code Review Activities. *arXiv preprint arXiv:2203.09095* (2022).

[29] Bo Lin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F Bissyandé. 2021. Automated Comment Update: How Far are We?. In *International Conference on Program Comprehension*. 36–46.

[30] 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. *arXiv preprint arXiv:1907.11692* (2019).

[31] Zhongxin Liu, Xin Xia, David Lo, Meng Yan, and Shanping Li. 2021. Just-In-Time Obsolete Comment Detection and Update. *IEEE Transactions on Software Engineering* (2021).

[32] 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. *arXiv preprint arXiv:2102.04664* (2021).

[33] Lara Martin, Prithviraj Ammanabrolu, Xinyu Wang, William Hancock, Shruti Singh, Brent Harrison, and Mark Riedl. 2018. Event representations for automated story generation with deep neural nets. In *AAAI Conference on Artificial Intelligence*. 868–875.

[34] 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 *International Conference on Software Engineering*. 336–347.

[35] Courtney Napoles, Keisuke Sakaguchi, Matt Post, and Joel Tetreault. 2015. Ground truth for grammatical error correction metrics. In *Annual Meeting of the Association for Computational Linguistics and International Joint Conference on Natural Language Processing*. 588–593.

[36] Graham Neubig, Makoto Morishita, and Satoshi Nakamura. 2015. Neural Reranking Improves Subjective Quality of Machine Translation: NAIST at WAT2015. In *Workshop on Asian Translation*. 35–41.

[37] 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 *International Conference on Software Engineering*. 2006–2018.

[38] Sheena Panthaplackel, Miltiadis Allamanis, and Marc Brockschmidt. 2021. Copy that! editing sequences by copying spans. In *AAAI Conference on Artificial Intelligence*. 13622–13630.

[39] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J Mooney. 2021. Deep just-in-time inconsistency detection between comments and source code. In *AAAI Conference on Artificial Intelligence*. 427–435.

[40] Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond Mooney. 2020. Learning to Update Natural Language Comments Based on Code Changes. In *Annual Meeting of the Association for Computational Linguistics*. 1853–1868.

[41] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Annual Meeting of the Association for Computational Linguistics*. 311–318.

[42] Karl Pichotta and Raymond Mooney. 2016. Learning statistical scripts with LSTM recurrent neural networks. In *AAAI Conference on Artificial Intelligence*. 2800–2806.

[43] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1 (2019), 9.

[44] 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.

[45] Ehud Reiter and Robert Dale. 1997. Building Applied Natural Language Generation Systems. *Natural Language Engineering* 3 (1997), 57–87.

[46] Alexander M Rush, Sumit Chopra, and Jason Weston. 2015. A Neural Attention Model for Abstractive Sentence Summarization. In *Empirical Methods in Natural Language Processing*. 379–389.

[47] Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. 2020. Learning to fix build errors with graph2diff neural networks. In *International Conference on Software Engineering Workshops*. 19–20.

[48] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *International Conference on Software Engineering*. 25–36.

[49] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *Transactions on Software Engineering* 28 (2019), 1–29.

[50] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using Pre-Trained Models to Boost Code Review Automation. In *International Conference on Software Engineering*. 2291–2302.

[51] Rosalia Tufano, Luca Pascarella, Michele Tufanoy, Denys Poshyvanykz, and Gabriele Bavota. 2021. Towards Automating Code Review Activities. In *International Conference on Software Engineering*. 163–174.

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 5998–6008.

[53] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. *Advances in Neural Information Processing Systems*.

[54] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021. SynCoBERT: Syntax-Guided Multi-Modal Contrastive Pre-Training for Code Representation. *arXiv preprint arXiv:2108.04556* (2021).

[55] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Empirical Methods in Natural Language Processing*. 8696–8708.

[56] Wei Xu, Courtney Napoles, Ellie Pavlick, Quanze Chen, and Chris Callison-Burch. 2016. Optimizing statistical machine translation for text simplification. *Transactions of the Association for Computational Linguistics* 4 (2016), 401–415.

[57] Ziyu Yao, Frank F. Xu, Pengcheng Yin, Huan Sun, and Graham Neubig. 2021. Learning Structural Edits via Incremental Tree Transformations. In *International Conference on Learning Representations*.

[58] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Annual Meeting of the Association for Computational Linguistics*. 440–450.

[59] Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L Gaunt. 2018. Learning to Represent Edits. In *International Conference on Learning Representations*.

[60] Qingyu Zhou, Nan Yang, Furu Wei, and Ming Zhou. 2018. Sequential copying networks. In *AAAI Conference on Artificial Intelligence*. 4987–4994.

[61] Zhi-Hua Zhou and Ming Li. 2005. Tri-training: Exploiting unlabeled data using three classifiers. *Transactions on knowledge and Data Engineering* 17 (2005), 1529–1541.