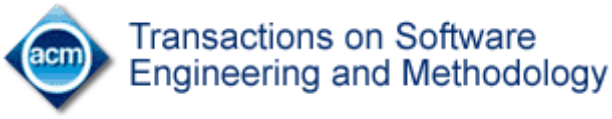


You Don’t Have to Say Where to Edit! Joint Learning to Localize and Edit Source Code

Journal:	*Transactions on Software Engineering and Methodology*
Manuscript ID	TOSEM-2023-0391
Manuscript Type:	Continuous Special Section: AI and SE
Date Submitted by the Author:	29-Oct-2023
Complete List of Authors:	Pian, Weiguo; University of Luxembourg Li, Yinghua; University of Luxembourg, Tian, Haoye; University of Luxembourg, SnT Sun, Tiezhu; University of Luxembourg, SnT Song, Yewei; University of Luxembourg Tang, Xunzhu; University of Luxembourg Habib, Andrew; University of Luxembourg Klein, Jacques; University of Luxembourg Bissyande, Tegawende; University of Luxembourg
Computing Classification Systems:	

To whom it may concern,

We are submitting our paper "You Don’t Have to Say Where to Edit! Joint Learning to Localize and Edit Source Code" to TOSEM under the "Continuous Special Section: AI and SE" manuscript type. The paper is an original contribution since the manuscript and any portion of it were previously never published in conferences nor journals. The following statements describe our novelty to satisfy the requirements of the manuscript type.

1. We propose jLED, a novel supervised learning approach designed to enable the practical application of code editing without the need for edit location. jLED leverages large-scale language models to uniformly localize and edit source code.

2. We conduct comprehensive experiments to evaluate the performance changes by employing different modalities for sequence-to-sequence editing baselines.

3. We collect a large-scale dataset of 77,044 edited code samples from two famous github projects – Linux and Wireshark, we extensively evaluate the effectiveness of jLED to localize and edit source code. The results demonstrate our tool jLED outperforms or achieves competitive performance when compared against the localizing and editing baselines, using five different large pre-trained code models trained with our pipeline.

4. To further evaluate the effectiveness of our proposed joint learning pipeline, we construct a two-stage localization-editing pipeline, in which a localization model and a editing model are trained separately. We conduct experiments for ablation study with two-stage pipeline, and experimental results further demonstrate the superiority of our proposed joint learning pipeline.

Sincerely,
Weiguo Pian, Yinghua Li, Haoye Tian, Tiezhu Sun, Yewei Song, Xunzhu Tang, Andrew Habib, Jacques Klein, and Tegawendé F. Bissyandé.

You Don't Have to Say Where to Edit! Joint Learning to Localize and Edit Source Code

WEIGUO PIAN[†] and YINGHUA LI[†], University of Luxembourg, Luxembourg
HAOYE TIAN[*], University of Luxembourg, Luxembourg
TIEZHU SUN, University of Luxembourg, Luxembourg
YEWEI SONG, University of Luxembourg, Luxembourg
XUNZHU TANG, University of Luxembourg, Luxembourg
ANDREW HABIB, University of Luxembourg, Luxembourg
JACQUES KLEIN, University of Luxembourg, Luxembourg
TEGAWENDÉ F. BISSYANDÉ, University of Luxembourg, Luxembourg

Learning to edit code automatically is becoming more and more feasible. Thanks to recent advances in Neural Machine Translation, various case studies are being investigated where patches are automatically produced and assessed either automatically (using test suites) or by developers themselves. An appealing setting remains when the developer must provide a natural language input of the requirement for the code change. A recent proof of concept in the literature showed that it is indeed feasible to translate these natural language requirements into code changes. As a state-of-the-art, this approach presents a significant constraint: it requires the developer to precisely indicate the code location (*i.e.*, code line) to be changed. In this work, we propose to address the challenge of generating code changes without precise location information. We consider this setting as realistic towards the practical adoption of NMT for code development. To that end, we develop a novel joint training approach for both localization and source code edition. Building a benchmark based on over 70k commits (patches and messages), we demonstrate that our JLED (**j**oint **L**ocalize and **ED**it) approach is effective. An ablation study further demonstrates the importance of our design choice in joint training.

CCS Concepts: • **Software and its engineering → Software verification and validation**; *Software defect analysis*; Software testing and debugging.

Additional Key Words and Phrases: Source Code Edition, Joint Learning, Fault Localization, Automated Programming, Neural Machine Translation

[†]Equal contribution.
[*]Corresponding author.

Authors' addresses: Weiguo Pian[†], weiguo.pian@uni.lu; Yinghua Li[†], yinghua.li@uni.lu, University of Luxembourg, Luxembourg; Haoye Tian, haoye.tian@uni.lu, University of Luxembourg, Luxembourg; Tiezhu Sun, tiezhu.sun@uni.lu, University of Luxembourg, Luxembourg; Yewei Song, yewei.song@uni.lu, University of Luxembourg, Luxembourg; Xunzhu Tang, xunzhu.tang@uni.lu, University of Luxembourg, Luxembourg; Andrew Habib, andrew.a.habib@gmail.com, University of Luxembourg, Luxembourg; Jacques Klein, jacques.klein@uni.lu, University of Luxembourg, Luxembourg; Tegawendé F. Bissyandé, tegawende.bissyande@uni.lu, University of Luxembourg, Luxembourg.

0004-5411/2023/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Weiguo Pian†, Yinghua Li†, Haoye Tian, Tiezhu Sun, Yewei Song, Xunzhu Tang, Andrew Habib, Jacques Klein, and Tegawendé F. Bissyandé. 2023. You Don't Have to Say Where to Edit! Joint Learning to Localize and Edit Source Code. *J. ACM* 1, 1 (October 2023), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Code editing [35, 41] is a critical and continuous activity in the realm of software development. As software systems expand in size and complexity, developers undertake a multitude of edits to maintain and enhance their functionality. These edits may include bug fixes [13, 27, 50], feature additions [33, 34, 36], or performance improvements [31]. Nevertheless, a significant portion of code editing across various projects is repetitive or similar in practice, which results in decreased efficiency in software development [42]. Therefore, researchers are motivated to devise automated approaches that can facilitate code editing by learning from historical examples [35, 36].

Approaches based on Neural Machine Translation (NMT) have demonstrated remarkable success in the realm of automated code editing. NMT is a type of machine learning approach that uses the sequence-to-sequence [46] architecture to predict the target sequence based on the source sequence of words or tokens, which is commonly used to translate sentences from one language to another, or to generate answers from questions. In the context of code editing, the process refers to the translation of source code to targeted code. Contrasting to traditional machine translation methods that translate words or phrases in isolation, NMT models the entire context of a sentence or even a paragraph to produce a nuanced translation [62]. Leveraging this capability, literatures [6, 30, 64] have successfully adapted NMT models to comprehend code semantics and generate more accurate and contextually relevant edits by leveraging multiple modalities of information relevant to code editing, such as the context, natural language guidance, test cases, etc.

Despite the achievements, NMT-based approaches face a substantial obstacle in real-world applications: NMT models require the location of the edit (e.g., the line of buggy code) as an input, which is often unavailable in practical scenarios. Consequently, NMT struggles to edit code effectively within a broad context without the knowledge of the exact edit location. Chakraborty*et al.* [6] investigated the contribution of different input modalities to the performance of their proposed NMT model. The findings indicate a considerable decline in performance when the edit location remains unknown to the NMT model. This however hinders the adoption of the NMT-based code edit approaches to practical scenarios.

This paper. We propose to jointly optimize the two loss functions of edit location and code edition in NMT models, towards producing an integrated approach to enable precise localization and edition of source code without the knowledge of exact edit locations. The main contributions are as follows:

❶ Our paper introduces JLED (**j**ointly **L**ocalize and **ED**it), a novel supervised learning approach designed to enable the practical application of code editing without the need for edit location. JLED leverages large-scale language models to uniformly localize and edit source code.
❷ We conduct comprehensive experiments to evaluate the performance changes by employing different modalities for sequence-to-sequence editing baselines.
❸ After collecting a large dataset of 77,044 edited code samples from two famous github projects – Linux and Wireshark, we extensively evaluate the effectiveness of JLED to localize and edit source code. The results demonstrate our tool JLED outperforms or achieves competitive performance when compared against the localizing and editing baselines, using five different large pre-trained code models trained with our pipeline.
❹ To further evaluate the effectiveness of our proposed joint learning pipeline, we construct a two-stage localization-editing pipeline, in which a localization model and a editing model are

trained separately. We conduct experiments for ablation study with two-stage pipeline, and experimental results further demonstrate the superiority of our proposed joint learning pipeline.

Availability. Our artifact, code, and dataset are publicly available at: https://anonymous.4open.science/r/Code_Edit_Joint_Learning-1F8D.

The remainder of this paper is presented as follows. Section 2 introduces the background of this work. In Section 3, we present our methodology with detailed explanations. Section 4 and 5 cover the experimental design and results. We provide discussions and related work in Section 6 and 7. Section 8 concludes this work.

2 BACKGROUND

2.1 Neural Machine Translation

Neural Machine Translation (NMT) has emerged as a promising approach in the field of machine translation, exhibiting well performance in automating language translation tasks [6]. By leveraging deep neural networks, NMT models are capable of learning and generating translations in an end-to-end manner, thereby overcoming the limitations of traditional statistical machine translation methods. At its core, NMT comprises two fundamental components: the encoder and the decoder, which work synergistically to facilitate the translation process. The encoder component plays a crucial role in comprehending and processing the input sentence, utilizing sophisticated neural architectures to generate a vector representation that encapsulates the underlying semantic meaning of the source text. This vector representation serves as a rich and comprehensive representation of the source sentence, enabling the subsequent translation process to leverage the encoded information effectively. The decoder component, on the other hand, capitalizes on the encoded input representation to sequentially generate the target sentence through a process of logical reasoning.

In recent years, the field of Software Engineering (SE) has witnessed a broad range of applications for NMT. Notably, NMT has found utility in areas such as Automatic Program Repair [20, 30], Program Synthesis [65], and Code Edit Generation [55, 56]. These applications capitalize on NMT's ability to comprehend and generate intricate patterns, making it a valuable tool for SE-related tasks. The integration of NMT in software engineering research highlights its potential to enhance various aspects of software development, offering new avenues for improving program understanding, code generation, and automated repair.

2.2 Transformer Model for Sequence Processing

Transformer model [58] has emerged as a highly influential and prominent model for sequence processing tasks within the field of natural language processing (NLP), leading to numerous state-of-the-art achievements [3, 61]. Prior to the advent of the Transformer, recurrent neural networks (RNNs) and their variants, such as long short-term memory (LSTM) and gated recurrent units (GRUs), were conventionally utilized for sequence processing tasks. While RNNs showcased commendable performance, they encountered challenges in parallelization and capturing long-range dependencies adequately. To overcome these limitations, the Transformer model introduced a novel self-attention mechanism, enabling selective attention to different parts of the input sequence through adaptive weighting. This mechanism played a pivotal role in capturing dependencies between distinct positions within the sequence, facilitating parallel processing of the input. During the token representation learning phase, the Transformer model learned to attend to all input tokens, transforming the sequence into a comprehensive graph where each token represented a node. The edge weights in this graph denoted the attention weights between tokens, which were learned based on the specific task at hand. Additionally, positional encoding was incorporated

into the Transformer model to encode the position of each token in the sequence, facilitating the learning of long-range dependencies. The Transformer's ability to reason about long-range dependencies has proven to be highly advantageous for various source code processing tasks, including code generation [47] and code summarization [2].

2.3 Transfer Learning for Source Code

Transfer learning [60] has emerged as a prominent research direction in the domain of software engineering (SE) due to its potential to address various SE tasks effectively. In SE, transfer learning involves the creation of task-agnostic representations of source code, which can be leveraged and repurposed across different tasks. One prevalent approach to obtain such task-agnostic representations entails pre-training models using a large corpus of source code. During the pre-training phase, the primary objective is to enhance the model's understanding of code or its ability to generate accurate code. By leveraging a substantial collection of source code, a pre-trained model is expected to encapsulate valuable code-related knowledge within its learnable parameters. Subsequently, these pre-trained models are fine-tuned to adapt to specific task objectives.

Several transformer-based encoder models have been developed to facilitate pre-training for comprehending source code. Notable examples include CodeBERT [11] and GraphCodeBERT [14]. CodeBERT [11] focuses on learning continuous representations of code snippets, enabling a deeper understanding of their structural and contextual aspects. GraphCodeBERT [14] incorporates graph neural networks to capture code dependencies and interactions, facilitating higher-level reasoning and tasks such as code completion and refactoring. In the context of code generation, two prominent models are CodeGPT [29] and PLBART [1]. CodeGPT pre-trains a transformer-based model specifically designed for sequentially generating general-purpose code. More recently, PLBART introduces a joint pre-training approach that encompasses both code understanding and code generation, utilizing denoising auto-encoding. PLBART consists of an encoder and a decoder, where the encoder is exposed to slightly perturbed code, while the decoder is responsible for producing code without such perturbations.

3 APPROACH

In this section, we delve into a detailed presentation of our proposed approach. Figure 1 illustrates the pipeline of our JLED. The overall pipeline is composed of three primary components: input pre-processing, model architecture, and the optimization objective. Note that, during the inference phase, the optimization objective is replaced by an output generation module. In the pre-processing stage, we concatenate different parts/modalities, converting them into a token sequence for input.

3.1 Input Modalities & Pre-processing

As we mentioned before, since we may not know the exact line-level location in which the line content should be edited, it is not in practice to provide the exact line-level location information as a part of the input of the model. Therefore, in our setting, the model can only take natural language guidance and code context (a multiple lines code fragment) as input. We use $\mathcal{G}$ and C to denote the natural language guidance and the code context respectively, and we follow the setting in [6] to use a special token <s> to split the two modalities $\mathcal{G}$ and C.

In the Pre-processing step, we then apply the tokenizer to tokenize the input into sequences of tokens. For the tokenizer, we follow Chakraborty et al. [6] and use the sentence-piece tokenizer [24] in all our experiments, which divides each token into sequences of subtokens. In our implementation, for different pre-trained models used in our training pipeline, we apply their original pre-trained sentence-piece tokenizer. Finally, the tokenized sequence is generated after the data pre-processing as the input of the model.

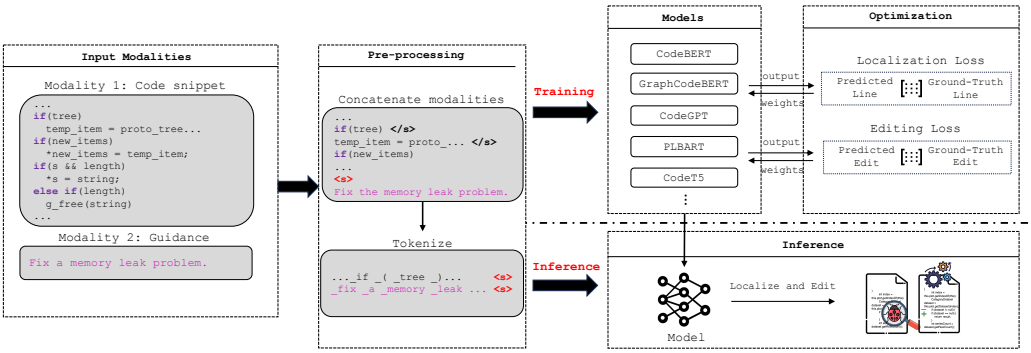


Fig. 1. Overview of our JLED pipeline.

3.2 Models

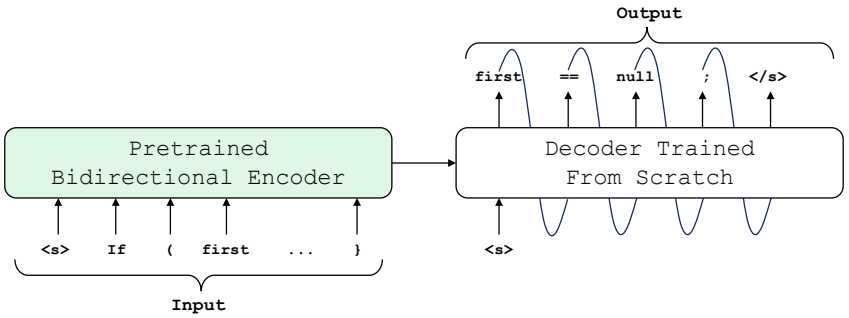
Recently, there has been a significant surge in interest of Transformer-based [58] code models [38, 39, 68] in fields of code representation learning and software engineering. These models, especially pre-trained large code models [1, 11, 14, 29, 59], attract lots of attention due to their superior performance and generalizability, proving to be highly advantageous for various research topics in software engineering [4, 17–19, 32, 49, 51, 53]. Inspired by this, we also apply the pre-trained large code models in our approach, which can be divided into three categories: encoder pre-trained code models, decoder pre-trained code models, and encoder-decoder pre-trained code models. Specifically, the representative encoder pre-trained code models include CodeBERT [11] and GraphCodeBERT [14], while the one of the most representative decoder pre-trained models is CodeGPT [29]. Recently, researchers also explored the ability of pre-trained encoder-decoder models in the field of code representation learning and proposed the pre-trained encoder-decoder code models PLBART [1] and CodeT5 [59]. In the rest of this subsection, we introduce the details of these pre-trained models' architecture.

The basic component of the large code models is the encoder-decoder architecture, a powerful sequence-to-sequence deep learning architecture, which has been widely used in text-to-text task, text-to-code task, code-to-text, and code-to-code (our task) tasks. Note that, in our experiments, all the models used in our proposed JLED are based on the encoder-decoder architecture, except CodeGPT which is a decoder-only model. We will discuss it later.

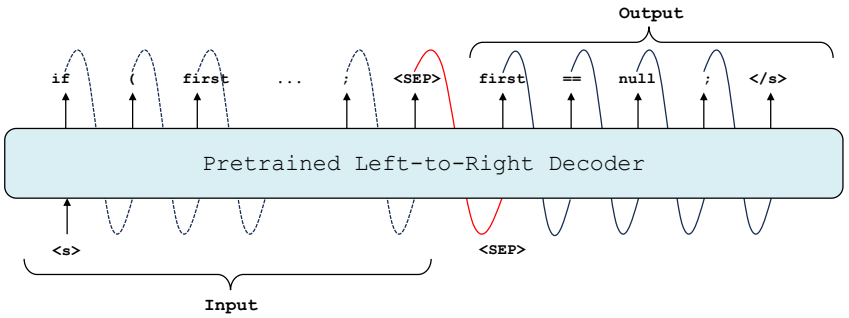
Encoder. Encoder is the first part of the encoder-decoder architecture, which is used for encoding the pre-processed input subtokens sequence (see last subsection for details) into the semantic feature space to obtain the representation of the input sequence. For an L^e-layers encoder model, the l-th layer's output feature can be denoted as:

$$\begin{aligned} \boldsymbol{X}_e^l &= \mathcal{F}_e^l(\boldsymbol{X}_e^{l-1}; \boldsymbol{W}_e^l), \\ s.t. \quad \boldsymbol{X}_e^{l-1} &= \{\boldsymbol{x}_1^{l-1}, \boldsymbol{x}_2^{l-1}, ..., \boldsymbol{x}_n^{l-1}\} \end{aligned} \tag{1}$$

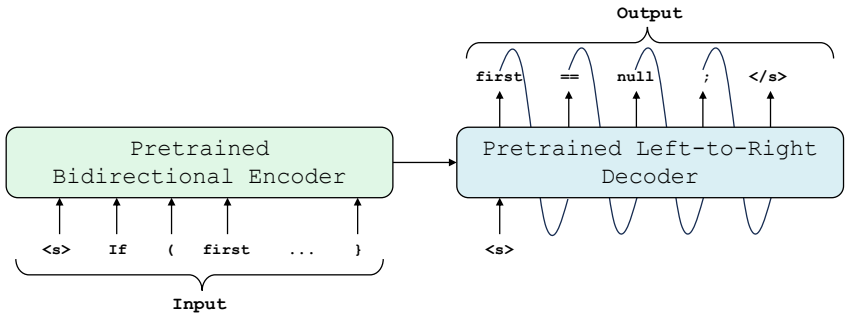
where $\boldsymbol{X}_e^{l-1} = \{\boldsymbol{x}_1^{l-1}, \boldsymbol{x}_2^{l-1}, ..., \boldsymbol{x}_n^{l-1}\}$ denotes the intermediate feature of the input subtokens sequence, and $\boldsymbol{x}_i^{l-1} \in \mathbb{R}^d$ and d are the intermediate representation of the i-th subtoken generated by the l-th encoder layer and the length of the intermediate representation of each subtoken, respectively. Note that, we use $\boldsymbol{X}_e^0 = \{\boldsymbol{x}_1^0, \boldsymbol{x}_2^0, ..., \boldsymbol{x}_n^0\}$ to represent the original input subtokens sequence. $\mathcal{F}_e^l$, parameterized by the learnable weights $\boldsymbol{W}_e^l$, is the l-th layer of the encoder model. For an L-layers encoder model, the final output of the encoder model can be denoted as $\boldsymbol{X}_e^L \in \mathbb{R}^{n \times d}$, which will be used as the input of the decoder model to generate the prediction (target sequence, code edits in



(a) Encoder pre-trained encoder-decoder models' architecture — Consists of bidirectional pre-trained encoder and a decoder trained from scratch.



(b) Decoder-only pre-trained models' architecture — One pre-trained single decoder processes the input and output sequentially from left to right.



(c) Joint encoder-decoder pre-trained models' architecture — Consists of pre-trained bidirectional encoder and pre-trained left to right decoder.

Fig. 2. Schematic diagram of the three types of pre-trained models: (a) Encoder pre-trained encoder-decoder models, (b) Decoder-only pre-trained models, and (c) Joint encoder-decoder pre-trained models.

our task), as well as the input of the localization branch of our proposed pipeline. In the rest of this paper, we simply use $\boldsymbol{Z}$ and $\boldsymbol{X}$ to denote $\boldsymbol{X}_e^L$ and $\boldsymbol{X}_e^0$ respectively, and use $\mathcal{F}_e$ and $\boldsymbol{W}_e$ to denote the entire encoder model and its trainable parameters.

Decoder. Decoder is another part of the encoder-decoder architecture, which aims to decode the representation generated by the encoder, to the target output subtokens sequence. In sequece-to-sequence tasks, the decoder generate subtokens sequentially using the encoder generated global representation and previous decoder generated subtokens. For the n-th subtoken's generation, the decoder takes previous generated subtokens and the encoder generated representation/hidden states as the input, to predict next subtoken. This process can be denoted as:

$$\begin{aligned} \boldsymbol{u}_n =& \mathcal{F}_d(\boldsymbol{U}_{n-1}, \boldsymbol{Z}; \boldsymbol{W}_d), \\ s.t.\ \ \boldsymbol{U}_{n-1} =& \{\boldsymbol{u}_0, \boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_{n-1}\}, \end{aligned} \tag{2}$$

where $\mathcal{F}_d$ denotes the decoder model with learnable parameters $\boldsymbol{W}_d$. $\boldsymbol{u}_i$ denotes the i-th output subtoken. Please note that, $\boldsymbol{u}_0$ is a special token that indicates the start of outputs generation.

Note that, for decoder-only models, *e.g.* CodeGPT [29], due to the lacking of encoder part, when generate the n-th subtoken $\boldsymbol{u}_n$, the model only takes $\boldsymbol{U}_{n-1}$ as input without the intermediate representation $\boldsymbol{Z}$ in the modeling process, which can be represented as:

$$\begin{aligned} \boldsymbol{u}_n =& \mathcal{F}_d(\boldsymbol{U}_{n-1}; \boldsymbol{W}_d), \\ s.t.\ \ \boldsymbol{U}_{n-1} =& \{\boldsymbol{u}_0, \boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_{n-1}\}, \end{aligned} \tag{3}$$

Based on above encoding and decoding processes, models inference with encoder-decoder architecture (encoder pre-trained models and encoder-decoder pre-trained models), *e.g.* CodeBERT [11], GraphCodeBERT [14], PLBART [1], and CodeT5 [59], in code editing task can be denoted as:

$$\begin{aligned} \boldsymbol{u}_n =& \mathcal{F}_d(\boldsymbol{U}_{n-1}, \boldsymbol{Z}; \boldsymbol{W}_d), \\ s.t.\ \ \boldsymbol{U}_{n-1} =& \{\boldsymbol{u}_0, \boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_{n-1}\},\ \ \boldsymbol{Z} = \mathcal{F}_e(\boldsymbol{X}; \boldsymbol{W}_e), \end{aligned} \tag{4}$$

Similarly, the code editing task in decoder-only pre-trained model, *e.g.* CodeGPT [29], can be expressed as:

$$\begin{aligned} \boldsymbol{u}_n =& \mathcal{F}_d([\boldsymbol{X}, \boldsymbol{U}_{n-1}]; \boldsymbol{W}_d), \\ s.t.\ \ \boldsymbol{U}_{n-1} =& \{\boldsymbol{u}_0, \boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_{n-1}\}, \end{aligned} \tag{5}$$

We use the pre-processed subtokens sequence (see last subsection for details) as the input of the model. In this subsection, we introduce the details of the models that can be used in our approach.

In summary, Equation 4 and 5 present the edits generation process in the code editing task. However, as we mentioned before, in real-world code editing, given a code chunk/snippet, we usually don't know the exact line location where we should edit. That is, the modality of *the content of the code line that to be edited*, is unknown. In this situation, since the model does not learn *where* to be edited in the given code chunk/snippet, as well as the content to be edited, existing standard end-to-end sequence-to-sequence training/fine-tuning strategy is inadequate for the enabling the model to generate precise code edits. To tackle this problem, we propose JLED, a joint training strategy to allow the model to learn to localize and edit simultaneously, in which the localization task can be used to facilitate editing generation process for unknown edit location. In the following subsection, we introduce the details of our proposed JLED.

3.3 Optimization

In this subsection, we introduce the details of our proposed joint training loss function for our JLED.

3.3.1 Editing Loss. The first component of our final loss function is the editing loss, serving as the optimization target to minimize the distance (loss score) between the generated edits (sequence of output subtokens) and the ground truth edits. The editing loss can be denoted as:

$$\mathcal{L}_{edit} = \mathbb{E}_{X\sim\mathcal{D}}\Big[\sum_{i=1}^{L}\mathcal{L}_{CE}(\boldsymbol{u}_i, \boldsymbol{y}_i)\Big] \tag{6}$$

where $\mathcal{D}$ denotes the training set, $\boldsymbol{y}_i$ is the i-th subtoken in the ground-truth subtoken sequence (ground truth edits) $\mathcal{Y}$, L is the maximum length of the output sequence, and $\mathcal{L}_{CE}$ is the cross-entropy loss function.

3.3.2 Localization Loss. Localization loss aims to guide the model to learn the exact line-level location that to be edited given a natural language description and associated code context. To predict the location line number, a predictor $\mathcal{P}(\cdot;\boldsymbol{\theta}_p)$ with learnable parameters $\boldsymbol{\theta}_p$ is added on the top layer of the model. The predictor takes the hidden state generated by the model from the input sequence as its input, to output the predicted location line number.

Specifically, in the encoder-decoder models, *e.g.* CodeBERT [11], GraphCodeBERT [14], PLBART [1], and CodeT5 [59], the predictor takes the encoder generated representation $\boldsymbol{Z}$ as the input and output the predicted location line number, which is used to calculate the localization loss with the ground truth location line number. This process can be denoted as:

$$\mathcal{L}_{loc.} = \mathbb{E}_{X\sim\mathcal{D}}\big[\mathcal{L}_{CE}(\mathcal{P}(\boldsymbol{Z};\boldsymbol{\theta}_p), \boldsymbol{l})\big] \tag{7}$$

where $\boldsymbol{l}$ denotes the ground location line number where to be edited.

Similarly, for decoder-only models, *e.g.* CodeGPT [29], the predictor takes the decoder generated hidden states of the input sequence as the its input to generate the line number prediction, which can be presented as:

$$\mathcal{L}_{loc.} = \mathbb{E}_{X\sim\mathcal{D}}\big[\mathcal{L}_{CE}(\mathcal{P}(\mathcal{F}_d(\boldsymbol{X};\boldsymbol{W}_d);\boldsymbol{\theta}_p), \boldsymbol{l})\big] \tag{8}$$

3.3.3 Joint Loss Function. Finally, the editing loss and the localization loss are combined as the joint loss function, which is denoted as:

$$\mathcal{L}_{joint} = \mathcal{L}_{edit} + \lambda\mathcal{L}_{loc.} \tag{9}$$

where λ is the hyperparameter to balance the values of the editing loss and localization loss. Incorporating a joint loss function, the model benefits from two optimization targets: localization and editing. The localization target enables the model to pinpoint the specific line number requiring edits, thus refining its focus during the edit generation process. Simultaneously, the editing training target further guides the model to accurately identify these line numbers. Owing to this synergistic training framework, the model is capable of generating highly accurate edits even when the exact line locations within the source code are unknown.

3.4 Inference

After the training process described in the previous sections, the trained model will be utilized to predict edits based on the pre-processed and tokenized subtoken sequence. Following a similar approach to the training process mentioned earlier, the model will generate another sequence of subtokens representing the predicted edits. These predicted edits will be used to calculate evaluation metrics and generate the post-editing code context. This process can also be expressed by Equations 4 and 5.

4 EXPERIMENTAL DESIGN

4.1 Dataset

To evaluate the effectiveness of our proposed approach, we collect a large-scale dataset with (1) before edited source code context, (2) natural language description, (3) line-level edits location (line number), and (4) the ground-truth edits. Our dataset is collected from two famous github projects – Linux and Wireshark. There are total 77,044 samples in the dataset. More specifically, 66,044, 5,500, and 5,500 samples for training, evaluation, and testing, respectively. For each sample in the dataset, it contains (1) a source code snippet with several lines as the code context, (2) a natural language description of the editing purpose as the natural language guidance, (3) a line number as the editing location, and (4) the ground-truth edits. Each of the sample is generated by using the `git` command to extract the code change, commit message, and the original code file before editing. Table 1 presents the statistic of the dataset.

Table 1. Statistic of the dataset

Split	Training	Validation	testing
Sample #	66,044	5,500	5,500

4.2 Data Preparation

For our collected dataset described in Section 4.1, we follow the pre-processing method described in Section 3.1 to pre-process each sample in the dataset by concatenating the source code context C and the associated natural language guidance $\mathcal{G}$ as an input data sample $[C$ <s> $\mathcal{G}]$ where <s> denotes the special token for splitting different modalities, and apply the tokenizer to tokenize each input data sample to generate the input subtoken sequence $\boldsymbol{X}$. In the modality of source code context C, we add another special token </s> to the end of each code line for splitting different lines of the source code context. In this way, the model is able to know how many lines in the source code context C as well as the exact start and end position of each line. And then, we extract each sample's ground-truth edit as the editing label $\mathcal{Y}$ and the location line to be edited as the localization label $\boldsymbol{l}$.

4.3 Training

After the preparation of each data point in the dataset, *i.e.* concatenating each sample's source code context C and the associated natural language guidance $\mathcal{G}$ into $[C$ <s> $\mathcal{G}]$ as the input, and extracting each sample's ground-truth edit and the location line to be edited as the editing label $\mathcal{Y}$ and the localization label $\boldsymbol{l}$. We apply the processed dataset to train and evaluate the models. To evaluate the performance of our proposed joint training under the scenario of editing source code using natural language guidance but without knowing the exact line location to be edited, we conduct experiments with various well-known and most used pre-trained models fine-tuned on our dataset, including encoder pre-trained encoder-decoder models, such as CodeBERT [11] and GraphCodeBERT [14], joint encoder-decoder pre-trained models, such as PLBART [1] and CodeT5 [59], and pre-trained decoder-only models, such as CodeGPT [29]. We train the model with our joint training pipeline, using the Adam [23] optimizer with the learning rate of 5e-5. As described before, both editing loss and localization loss are implemented by the cross-entropy loss function. We train each model to convergence, and use the beam search to generate output edits during inference (validation and testing). For all models, we set the balance hyperparameter λ to 0.1 during training. We implement the training and inference pipeline with Pytorch [37], and use the pre-trained parameters from Hugging Face.

4.4 Evaluation Metric

We use the BLEU score and Top-1 accuracy as the evaluation metric to evaluate the performance of the editing results. For Top-1 accuracy of the editing results, we follow the settings in the previous work [6], in which the beam size is set to 5, and then, *only the generated edits that perfectly match the ground-truth edits are correct.* This setting allows the most stringent metric for evaluation [6]. For the evaluation of the localization results, we also apply the Top-1 accuracy and the Top-5 accuracy as the evaluation metric. For the Top-1 accuracy in localization, the line number with the highest probability score in the line-aware probability distribution is considered as the predicted localization result to match the ground-truth localization label $\boldsymbol{l}$. For the Top-5 accuracy in localization, the prediction is considered as correct if any of our model's Top-5 highest probability prediction match with the ground-truth localization label $\boldsymbol{l}$.

4.5 Research Questions

Different from previous code editing approach, *i.e.*, MODIT [6], we consider a more practical code editing scenario, in which the exact location to be edited is unknown (the first modality in the input of MODIT [6]). To evaluate the performance of the existing standard sequence-to-sequence training with multi-modalities, *i.e.* training pipeline in the paper of MODIT [6], we firstly conduct experiments by training different large pre-trained code models on this state-of-the-art multi-modalities training pipeline [6] under our considered more practical setting, *i.e.*, only with the modalities of the source code context (C) and the natural language guidance ($\mathcal{G}$). And then, we conduct experiments by training different large pre-trained code models with our proposed joint training pipeline. Our research questions are as follows:

RQ-1. How do the models perform when trained on standard sequence-to-sequence multi-modalities training pipeline with only modalities of the source code context and the natural language guidance?

RQ-2. How does our joint training pipeline (jLED) perform compared to the baseline?

RQ-3. How does our joint training pipeline perform compared to the two-stage localization-editing pipeline?

5 EXPERIMENTAL RESULTS

✍ **RQ-1** ▶ *How do the models perform when trained on standard sequence-to-sequence multi-modalities training pipeline with only modalities of the source code context and the natural language guidance?* ◀

5.1 Experimental Setup for RQ-1

In our first research question, we aim to evaluate the performance of the standard sequence-to-sequence training (*i.e.*, only using the editing loss in Equation 6 as the training loss function) with multi-modalities (source code context $\mathcal{S}$ and natural language guidance $\mathcal{G}$). We name these trained models as baselines. To evaluate the performance of the baselines, we carefully choose the most representive and common pre-trained code models, including encoder pre-trained encoder-decoder models, decoder-only pre-trained models, and joint encoder-decoder pre-trained models. For the encoder pre-trained encoder-decoder models, we select CodeBERT [11] and GraphCodeBERT [14] in our pipeline. For the decoder-only pre-trained models, we train a CodeGPT [29] model. And for the joint encoder-decoder pre-trained models, we apply the most used PLBART [1] and CodeT5 [59] as the model to be trained in our approach. The description of the chosen models are as follows:

- **CodeBERT:** CodeBERT is a pre-trained code model based on the BERT architecture [8]. CodeBERT is pre-trained on a large-scale source code dataset using the pre-training scheme

of RoBERTa [28]. CodeBERT is the first large pre-trained NL-PL model for both natural language and source code modeling.

- **GraphCodeBERT:** GraphCodeBERT [14] is a large pre-trained code model based on the BERT architecture [8]. Different from CodeBERT that uses the pre-training strategy of RoBERTa [28] only on context information (masked language modeling), GraphCodeBERT also applies pre-training approaches on the data-flow graph, *i.e.*, cross code-graph variable-alignment and data flow edge prediction.
- **CodeGPT:** CodeGPT is a decoder-only source code model, which is based on the GPT architecture and pre-trained on the source code context. Similar to the GPT, CodeGPT is also trained with the autoregressive manner.
- **PLBART:** PLBART is a joint encoder-decoder pre-trained model based on the BART [25] architecture, and pre-trained with the denoising sequence-to-sequence strategy.
- **CodeT5:** CodeT5 is a joint encoder-decoder pre-trained model with the same model architecture of T5 [40], and pre-trained using the text-to-text training strategy as described in T5 [40].

We train the above described chose baselines on the training set of our dataset, and evaluate the performance of them. During the training process, we pre-process the data using the same method described in Section 3.1 and 4.2 and generate the input data as $[C$ <s> $\mathcal{G}]$, where <s> is the special token for splitting different modalities, C and $\mathcal{G}$ denote the source code context information and the natural language guidance respectively. For the baselines, as described before, they are the existing standard sequence-to-sequence multi-modalities training pipeline for source code editing, therefore, their training objective only contains a editing loss (Equation 6). Further, we also conduct experiments by training models under the setting in previous work of multi-modalities training for source code editing [6], in which the content of the code line to be edited is also included in the input as another modality. By this way, the input becomes $[\mathcal{E}$ <s> C <s> $\mathcal{G}]$, where $\mathcal{E}$ denotes the content of the code line to be edited. The experimental results of three fully input modalities can be seen as the *upper bound* of the results of with only source code context and natural language guidance modalities.

Table 2. Experimental results of different models trained with all three modalities ($[\mathcal{E}$ <s> C <s> $\mathcal{G}]$) and with only the modalities of source code context (C) and natural language guidance ($\mathcal{G}$).

Model	Modalities	BLEU	Top-1 Acc.
CodeBERT	$[C$ <s> $\mathcal{G}]$	50.32	41.95
	$[\mathcal{E}$ <s> C <s> $\mathcal{G}]$	65.77	53.24
GraphCodeBERT	$[C$ <s> $\mathcal{G}]$	52.58	43.38
	$[\mathcal{E}$ <s> C <s> $\mathcal{G}]$	66.00	53.98
CodeGPT	$[C$ <s> $\mathcal{G}]$	44.25	38.58
	$[\mathcal{E}$ <s> C <s> $\mathcal{G}]$	64.50	53.00
PLBART	$[C$ <s> $\mathcal{G}]$	52.20	45.45
	$[\mathcal{E}$ <s> C <s> $\mathcal{G}]$	67.89	57.36
CodeT5	$[C$ <s> $\mathcal{G}]$	56.85	50.87
	$[\mathcal{E}$ <s> C <s> $\mathcal{G}]$	68.41	59.75

5.2 Experimental Results for RQ-1

The experimental results for RQ-1 are shown in Table 2, where we present the BLEU score and the Top-1 accuracy of different models' results on the testing set with input modalities of $[\mathcal{C}$ <s> $\mathcal{G}]$ and $[\mathcal{E}$ <s> $\mathcal{C}$ <s> $\mathcal{G}]$ respectively. From the table, we can see that all the models' performance drops significantly after removing the content of the line to be edited (i.e., after removing $\mathcal{E}$). Specifically, compare to training with all three modalities, the CodeBERT drops 15.45 and 11.29 for BLEU score and Top-1 accuracy respectively. For GraphCodeBERT, the results of trained with $[\mathcal{C}$ <s> $\mathcal{G}]$ decrease by 13.42 and 10.60 for BLEU and Top-1 accuracy respectively, compared to training with the modalities of $[\mathcal{E}$ <s> $\mathcal{C}$ <s> $\mathcal{G}]$. For the decoder-only model CodeGPT, it drops by 20.25 and 14.42 for BLEU score and Top-1 accuracy respectively when trained without the modality of the content of the line to be edited $\mathcal{E}$. For the joint encoder-decoder pre-trained models PLBART and CodeT5, compare to training with all three modalities, when trained with only the modalities of source code context and natural language guidance, the BLEU scores of them decrease by 15.69 and 11.56 respectively, and the Top-1 accuracy of them drops by 11.91 and 8.88 respectively.

This phenomenon can be explained as follows: (i) The models do not know where to edit in the source code context, therefore, compare to the upper bound, the models trained with $[\mathcal{C}$ <s> $\mathcal{G}]$ lacks more information that would help generate the exact edits. (ii) During the training process, the models do not learn how to localize the line-level location where it is to be edited.

Based on these results, we can conclude that the models do not perform well when trained with only the modalities of source code context ($\mathcal{C}$) and natural language guidance ($\mathcal{G}$), since the models cannot capture the information that is associated to the ground-truth edits directly, and the models also do not learn knowledge and abilities to help themselves localize the line-level location that is to be edited from the given source code context ($\mathcal{C}$) and the natural language guidance ($\mathcal{G}$). Therefore, the answer to the RQ-1 appears:

> ✍ **Answer 1** ► *Under a more practical setting, in which only the source code context $\mathcal{C}$ and the natural language guidance $\mathcal{G}$ can be used to generate edits, standard sequence-to-sequence multi-modalities training pipeline is not enough for models to learn to generate accurate edits, since no more additional knowledge and abilities have been learned to help the model localize and edit exact code line.* ◄

To improve the performance of the models under this more practical scenario, in which only the source code context ($\mathcal{C}$) and the natural language guidance ($\mathcal{G}$) are available, we propose to train the models with the joint optimization target to enable the model learning both editing and localization abilities. To evaluate the effectiveness of our proposed approach, we investigate the next research question:

> ✍ **RQ-2** ► *How does our JLED perform compared to the baseline?* ◄

5.3 Experimental Setup for RQ-2

In the second research question, we aim to evaluate the performance of our proposed JLED. In the experiments of RQ-2, we select same pre-trained models used in the experiments of RQ-1, that are CodeBERT [11], GraphCodeBERT [14], CodeGPT [29], PLBART [1], and CodeT5 [59]. For the description of these models, please ref Section 5.1 for details. For each model, we train two baselines for it, which are the editing baseline and localization baseline respectively. More specifically, the editing baseline takes the modalities of source code context ($\mathcal{C}$) and natural language guidance ($\mathcal{G}$) as the input ($[\mathcal{C}$ <s> $\mathcal{G}]$), and uses the editing loss in Equation 6 as the loss function to train

Table 3. Experimental results of different models trained with only the modalities of source code context (C) and natural language guidance ($\mathcal{G}$) as input. All the models use only editing loss to train the editing baseline models, use only localization loss to train the localization baseline models, and use our joint loss function to train models for joint localization and editing.

Model	Modality	Training target	Editing results		Localization results	
			BLEU	Top-1 Acc.	Top-1 Acc.	Top-5 Acc.
CodeBERT	$[C$ <s> $\mathcal{G}]$	Editing Baseline	50.32	41.95	-	-
		Localization Baseline	-	-	62.09	**79.78**
		JLED (Ours)	**55.15**	**46.58**	**62.62**	78.76
GraphCodeBERT	$[C$ <s> $\mathcal{G}]$	Editing Baseline	52.58	43.38	-	-
		Localization Baseline	-	-	74.91	88.24
		JLED (Ours)	**55.71**	**47.20**	**74.96**	**89.34**
CodeGPT	$[C$ <s> $\mathcal{G}]$	Editing Baseline	44.25	38.58	-	-
		Localization Baseline	-	-	53.75	84.80
		JLED (Ours)	**49.90**	**42.71**	**62.49**	**85.28**
PLBART	$[C$ <s> $\mathcal{G}]$	Editing Baseline	52.20	45.45	-	-
		Localization Baseline	-	-	66.85	83.89
		JLED (Ours)	**54.78**	**48.84**	**70.09**	**86.62**
CodeT5	$[C$ <s> $\mathcal{G}]$	Editing Baseline	56.85	50.87	-	-
		Localization Baseline	-	-	**77.29**	88.93
		JLED (Ours)	**61.08**	**55.20**	77.07	**89.33**

the model. During inference, the editing baseline can generate the predicted edit, therefore, the evaluation metrics BLEU score and Top-1 accuracy can be used to evaluate the performance of the model. For the localization baseline, it also takes $[C$ <s> $\mathcal{G}]$ as the input, and uses the localization loss in Equation 7 (for CodeBERT, GraphCodeBERT, PLBART, and CodeT5) or Equation 8 (for CodeGPT) as the loss function to train the model. During the inference process of localization baselines, models output the predicted line-level position to be edited, *i.e.*, predicted line number. And then, we use the Top-1 and Top-5 accuracy to evaluate the predicted line-level position.

5.4 Experimental results for RQ-2

The experimental results for RQ-2 are presented in Table 3, where we show the BLEU score and the Top-1 accuracy of models' editing baselines and JLED (ours), and we also show the Top-1 and Top-5 accuracy of different models' localization baselines and JLED (ours). As described before, in all the experiments for RQ-2, all the models' editing baselines, localization baselines, and JLED take the modalities of $[C$ <s> $\mathcal{G}]$ as input.

Our experimental results presented in Table 3 offer a wealth of insights into the performance of our approach, JLED, compared to various baselines across multiple metrics. Most notably, JLED consistently outshines all the editing baselines in both BLEU score and Top-1 accuracy, irrespective of the underlying model architecture.

Specifically, when integrated with CodeBERT, JLED yields a significant uplift of **4.83** and **4.63** in BLEU score and Top-1 accuracy, respectively. GraphCodeBERT's performance also receives considerable boosts of **3.13** and **3.82** in BLEU and Top-1 metrics, reinforcing JLED's adaptability across different coding paradigms. The improvements are not merely incremental but substantial, highlighting JLED's generalization capabilities across different models.

Similarly, our JLED improves editing baseline of CodeGPT by **5.65** and **4.13** for BLEU score and Top-1 accuracy. For PLBART, JLED also boost the editing baseline by **5.65** and **4.13** for both metrics,

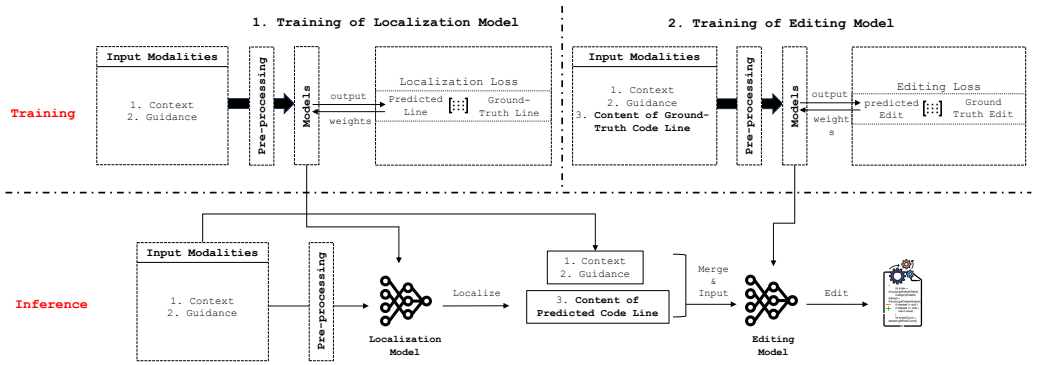


Fig. 3. Overview of the two-stage localization-editing pipeline.

further signifying its cross-model efficacy. CodeT5's performance also escalates, with gains of **5.23** and **4.33** in BLEU score and Top-1 accuracy, underscoring the versatility of JLED in adapting to varied model architectures and optimization landscapes.

On the other hand, with the assistance of the editing loss, the model can also localize the editing position more precisely compared to the localization baseline which is trained with only the localization loss. Specifically, compared to the localization baselines, the Top-1 accuracy improves by 0.53, 0.05, 8.74, and 3.24 for CodeBERT, GraphCodeBERT, CodeGPT, and PLBART respectively. For the Top-5 accuracy, the GraphCodeBERT, CodeGPT, PLBART, and CodeT5 outperform their localization baselines by 1.10, 0.48, 2.73, and 0.40, respectively. This suggests that JLED doesn't merely generate more precise edits, but also benefits the localization process, which is crucial for practical implementation.

In summary, these results demonstrate that our JLED enables models to acquire greater knowledge and abilities in localizing editing locations within the code context. This acquired knowledge and these abilities help the models focus on more precise potential editing locations when generating edits, leading to the production of more accurate edits. Additionally, learning to generate edits can also help the models learn to localize the editing position line more accurately. This also substantiate JLED's superior performance and adaptability across different model architectures and evaluation metrics. Whether in terms of edit quality or location precision, JLED consistently advances the state-of-the-art, making it a robust solution for automated code editing tasks.

Therefore, based on these experimental results and findings, we can conclude the answer to the RQ-2 as:

> ✍ **Answer 2** ► *Compared to the baselines, our JLED pipeline enables the models to learn additional knowledge and abilities regarding to localizing the editing location. Therefore, given only the modalities of source code context C and natural language guidance G, the models, which are trained by our JLED pipeline, can pay more attention on the potential editing location, so that yielding more precise and location-related edits.* ◄

> ✍ **RQ-3** ► *How does our joint training pipeline perform compared to the two-stage localization-editing pipeline?* ◄

5.5 Experimental Setup for RQ-3

Based on the experiments of RQ-2, we prove that our proposed joint training approach performs better than the baseline, since it enables the models to learn more knowledge and abilities in

localizing editing locations with the additional localization loss. This also proves that it is crucial for models to know locations before editing. However, there also has another manner that can also provide models some location information (predicted locations) except the joint training manner, that is, the two-stage localization-editing pipeline (we simply call it two-stage pipeline in the rest of this paper). The overview of the two-stage pipeline is shown in Figure 3. During the training process of the two-stage pipeline, we first train the model with the modalities of source code context ($\mathcal{C}$) and natural language guidance ($\mathcal{G}$) as input ($[\mathcal{C}$ <s> $\mathcal{G}]$), which is trained using only the localization loss (Equation 7 or 8) as the optimization target (same with the localization baseline in RQ-2). We name it as the localization model in the two-stage pipeline. And then, we train another model with the input of $[\mathcal{E}_{GT}$ <s> $\mathcal{C}$ <s> $\mathcal{G}]$, where $\mathcal{E}_{GT}$ denotes the content of the ground-truth code line to be edited. This model is named as editing model in the two-stage pipeline, optimized using the editing loss (Equation 6). During inference, given a sample with the modalities of code context ($\mathcal{C}$) and natural language guidance ($\mathcal{G}$), the trained localization model is used to predict the line-level location, which can be used to inquire the corresponding line-level content in the code context $\mathcal{C}$. We use $\mathcal{E}_{pred}$ to denote the inquired line-level content. After that, we concatenate the predicted line-level content to be edited ($\mathcal{E}_{pred}$), the source code context ($\mathcal{C}$), and the natural language guidance ($\mathcal{G}$) as the multimodal input $[\mathcal{E}_{pred}$ <s> $\mathcal{C}$ <s> $\mathcal{G}]$ of the trained editing model, to generate the predicted edits. Finally, we use the BLEU score and the Top-1 accuracy as the evaluation metrics for the generated edits. For the choice of the localization and editing models, we use the same settings in the experiments of RQ-1 and RQ-2, *i.e.*, CodeBERT [11], GraphCodeBERT [14], CodeGPT [29], PLBART [1], and CodeT5 [59].

Table 4. Experimental results of different models trained with our joint training approach and the two-stage approach.

Model	Approach	BLEU	Top-1 Acc.
CodeBERT	Two-stage	47.95	41.65
	JLED (Ours)	**55.15**	**46.58**
GraphCodeBERT	Two-stage	45.62	37.04
	JLED (Ours)	**55.71**	**47.20**
CodeGPT	Two-stage	40.80	36.65
	JLED (Ours)	**49.90**	**42.71**
PLBART	Two-stage	50.59	45.42
	JLED (Ours)	**54.78**	**48.84**
CodeT5	Two-stage	32.96	31.51
	JLED (Ours)	**61.08**	**55.20**

5.6 Experimental results for RQ-3

The experiments results of RQ-3 are presented in Table 4, where we show the evaluation results the two-stage training approach and our joint training approach regarding to CodeBERT, GraphCodeBERT, CodeGPT, PLBART, and CodeT5 respectively. From the table, we can see that, for all the models, our proposed joint training outperforms the two-stage training approach significantly. Specifically, for CodeBERT, our approach outperforms the two-stage method by **7.20** and **4.93** for BLEU score and Top-1 accuracy respectively. For GraphCodeBERT, our approach improves the two-stage approach by **10.09** and **10.16** for BLEU score and Top-1 accuracy. For CodeGPT, compared to the two-stage approach, our joint training has the improvement of **9.10** and **6.06** for BLEU score

and Top-1 accuracy respectively. For PLBART, our approach improves the two-stage method by **4.19** and **3.42** for BLEU and Top-1 accuracy respectively. For CodeT5, compared to the two-stage method, our proposed joint training approach outperforms it by **28.12** and **23.69** respectively. These experimental results demonstrate the superiority and the significant improvement of our proposed joint training approach over the two-stage method, so that we can conclude that our joint training is a better choice compared to the two-stage approach. The answer to RQ-3 can be denoted as:

✍ **Answer 3►** *The two-stage method, the editing performance of which is heavily rely on the localization model's predicted line-level location, which means that, when the predicted line-level location is incorrect, the generated edits in the next stage has a high probability of being wrong – even the input content of location to be edited ($\mathcal{E}$) is perfect, the generated edits are not 100% correct, let alone the input location is wrong. Compared to it, our joint training pipeline does not totally rely on the predicted localization results to generate the edits, so that the models could pay some attention to the correct code lines, even they are not with the highest probability scores. Based these experimental results, we conclude that our joint training pipeline is a better choice compared to the two-stage method.* ◄

6 DISCUSSION

6.1 Threats to Validity

The internal threat to validity lies in the bias of the characteristic of the model in our pipeline. To reduce this threats, we conduct experiments with different models for both baselines and our proposed pipeline. Further, we consider that it is not fair to compare different approaches implemented by different models, *e.g.*, compare our joint training approach implemented by CodeBERT to the CodeT5-based baseline, therefore, for fair comparison, we only compare different approaches within the same model.

Threats of external validity refer to the dataset used for the experiment. To reduce this threat, we construct a well-established dataset with high-quality data samples for training, validating, and testing our approach and the baselines.

6.2 Limitations

Our approach mainly focuses on the single line source code editing. However, when using the binary cross-entropy loss to replace the current standard cross-entropy loss, the localization parts of our approach can also handle multi-line source code editing by setting a threshold to filter multiple results as the predicted lines.

7 RELATED WORK

7.1 Automatic Code Change

Thanks to the repetitiveness of code editing, researchers have proposed to automate several code change tasks in the field of software engineering. One research direction aims to refactor existing code without changing its functionality [12, 22, 33]. For example, Meng *et al.* [33] introduced RASE, a highly advanced automated refactoring tool designed to eliminate redundancy in software code through clone removal. The evaluation showed that RASE successfully removes clones in a significant number of method pairs and groups with systematic edits, indicating the increased applicability of automated refactoring based on these edits. Khatchadourian *et al.* [22] transformed legacy Java code to leverage the new enumeration construct, improving type safety, code comprehension, simplicity, and eliminating brittleness issues. It employs an interprocedural type inferencing algorithm

and has been evaluated on 17 Java benchmarks. Other research direction addresses the completion or suggestion codes automatically [26, 43, 47, 48]. Svyatkovskiy *et al.* [47] proposed IntelliCode Compose, a versatile code completion tool capable of generating syntactically correct code sequences and entire lines in multiple programming languages. Leveraging a generative transformer model trained on extensive source code, IntelliCode Compose achieves high edit similarity and low perplexity for edit-time completion suggestions in Visual Studio Code IDE and Azure Notebook.

Another direction that is widely recognized as significant is the automation of bug detection and correction. With recent advances in deep learning, researchers proposed to use of NMT architecture or pre-trained encoder or decoder for program repair [5, 6, 15, 20, 45, 52, 64]. Our study is related to the last thrust. The evaluations conducted on the tasks showed promising results in automatic code change. Nevertheless, we argue that previous studies often assume the availability of a perfect edit location, such as fault location in program repair, which is not typically the case in real-world scenarios. Our objective is to fill this gap by jointly learning the localization and editing of source code.

7.2 Code Change Modeling

The code change modeling plays a crucial role in code-related tasks [7, 9–11, 16]. To learn distributed representations, Hoang *et al.* [16] utilized deep learning models to create CC2Vec, an approach of representing patches through sequence learning on code change. CC2Vec was evaluated as effective as the state of the arts on three patch tasks: generating patch descriptions, identifying bug-fixing patches, and just-in-time defect prediction. Similarly, CoDiSum [63] is a token-based approach to patch representation, particularly useful in generating patch descriptions. Moreover, Tufano *et al.* [55, 56] investigated the usage of NMT for general-purpose code changes learning.

The utility and adaptability of large-scale language models (LLMs) extend beyond natural language processing tasks. The BERT architecture [8], for instance, encodes both the left and right context of a token, making it adapted for the tasks of interpretation and generation of code that often relies heavily on the surrounding context. Several studies have revealed the efficacy of these models in handling source code. CodeBERT [11] and GraphCodeBERT [14], both derivatives of the BERT architecture, have shown considerable promise in understanding code semantics. CodeBERT is designed to tackle programming tasks by learning from bilingual data, including natural language and code, while GraphCodeBERT incorporates structural information from code into the pre-training process for better understanding the dependencies within the code. Furthermore, PLBART [1] and CodeT5 [59] have demonstrated substantial effectiveness in code translation and summarization tasks. PLBART, an encoder-decoder model, is specifically designed for programming language tasks. It leverages a large-scale bilingual corpus to learn a unified representation that captures the semantics of code. CodeT5, yet another encoder-decoder model, extend the T5 model by incorporating a code tokenizer and a new pre-training objective, making it successful in the task of code defect detection and clone detection.

Because of the outstanding representation ability of these pre-trained models on source code, they have also been widely adapted to various code change tasks such as program repair [49, 50, 53, 66], commit message generation [21, 44, 54] or code recommendation [57, 67]. In our proposed methodology, we initially harness the power of pre-trained LLMs as a fundamental architecture to represent both code and natural language. Subsequently, the weights of the LLMs are jointly optimized to learn to localize and repair bugs effectively. This approach allows the model to continually evolve and adapt, thereby enhancing its capability to perform code change tasks.

8 CONCLUSION

In this paper, we investigate the performance of existing standard sequence-to-sequence multi-modal learning for code editing under a more practical situation, in which the precise line-level location information is unknown or unavailable. Through comprehensive experiments, we have confirmed that the models are not able to generate precise edits after training without exact line-level location information. To tackle this challenge, we proposed JLED (**j**ointly **L**ocalize and **ED**it), a training pipeline to jointly learn to localize the edit buggy code simultaneously, which enables models to learn additional knowledge and abilities regarding to the line-level localization while learning to edit. We conduct experiments using our proposed JLED, and the experimental results show that our approach not only generates more precise edits, but also predicts more accurate line-level locations than the considered baselines. Moreover, to evaluate the effectiveness of our joint learning pipeline against other localization and editing alternatives, we construct a two-stage localization-editing pipeline, in which a localization model and a editing model are trained separately. Experimental results demonstrate the superiority of our JLED over this two-stage manner.

ACKNOWLEDGEMENTS

This work is supported by the NATURAL project, which has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant No. 949014).

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] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653* (2020).

[3] Wissam Antoun, Fady Baly, and Hazem Hajj. 2020. Arabert: Transformer-based model for arabic language understanding. *arXiv preprint arXiv:2003.00104* (2020).

[4] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T Devanbu, and Baishakhi Ray. 2022. NatGen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 18–30.

[5] 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.

[6] 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.

[7] Daniel DeFreez, Aditya V Thakur, and Cindy Rubio-González. 2018. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 423–433.

[8] 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*. 4171–4186. https://doi.org/10.18653/v1/n19-1423

[9] Jinhao Dong, Yiling Lou, Qihao Zhu, Zeyu Sun, Zhilin Li, Wenjie Zhang, and Dan Hao. 2022. FIRA: Fine-Grained Graph-Based Code Change Representation for Automated Commit Message Generation. (2022).

[10] Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. CodeTrans: Towards Cracking the Language of Silicon's Code Through Self-Supervised Deep Learning and High Performance Computing. *arXiv preprint arXiv:2104.02443* (2021).

[11] 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.

[12] Xi Ge and Emerson Murphy-Hill. 2014. Manual refactoring changes with automated refactoring validation. In *Proceedings of the 36th International Conference on Software Engineering*. 1095–1105.
[13] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.
[14] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations (ICLR)*.
[15] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*.
[16] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 518–529.
[17] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* 25 (2020), 2179–2217.
[18] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*. 1219–1231.
[19] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. *arXiv preprint arXiv:2302.05020* (2023).
[20] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.
[21] Tae-Hwan Jung. 2021. Commitbert: Commit message generation using pre-trained programming language model. *arXiv preprint arXiv:2105.14242* (2021).
[22] Raffi Khatchadourian. 2017. Automated refactoring of legacy Java software to enumerated types. *Automated Software Engineering* 24, 4 (2017), 757–787.
[23] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*, Yoshua Bengio and Yann LeCun (Eds.).
[24] Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 66–71.
[25] 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.
[26] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2018. Code completion with neural attention and pointer networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. 4159–25.
[27] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 31–42.
[28] 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).
[29] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 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)*.
[30] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.
[31] Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. 2023. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867* (2023).
[32] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 505–509.
[33] Na Meng, Lisa Hua, Miryung Kim, and Kathryn S McKinley. 2015. Does automated refactoring obviate systematic editing?. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 392–402.

[34] Na Meng, Miryung Kim, and Kathryn S McKinley. 2011. Sydit: Creating and applying a program transformation from an example. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 440–443.

[35] Na Meng, Miryung Kim, and Kathryn S McKinley. 2011. Systematic editing: generating program transformations from an example. *ACM SIGPLAN Notices* 46, 6 (2011), 329–342.

[36] Na Meng, Miryung Kim, and Kathryn S McKinley. 2013. LASE: locating and applying systematic edits by learning from examples. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 502–511.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[38] Han Peng, Ge Li, Wenhan Wang, Yunfei Zhao, and Zhi Jin. 2021. Integrating tree path in transformer for code representation. *Advances in Neural Information Processing Systems* 34 (2021), 9343–9354.

[39] Weiguo Pian, Hanyu Peng, Xunzhu Tang, Tiezhu Sun, Haoye Tian, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. 2023. MetaTPTrans: A meta learning approach for multilingual code representation learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5239–5247.

[40] 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. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[41] Sarah Rastkar and Gail C Murphy. 2013. Why did this code change?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 1193–1196.

[42] Baishakhi Ray, Meiyappan Nagappan, Christian Bird, Nachiappan Nagappan, and Thomas Zimmermann. 2015. The uniqueness of changes: Characteristics and applications. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 34–44.

[43] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*. 419–428.

[44] Ensheng Shi, Yanlin Wang, Wei Tao, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2022. RACE: Retrieval-Augmented Commit Message Generation. *arXiv preprint arXiv:2203.02700* (2022).

[45] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653* (2023).

[46] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).

[47] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.

[48] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2727–2735.

[49] Haoye Tian, Yinghua Li, Weiguo Pian, Abdoul Kader Kabore, Kui Liu, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. 2022. Predicting Patch Correctness Based on the Similarity of Failing Test Cases. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–30.

[50] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F Bissyandé. 2020. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 981–992.

[51] Haoye Tian, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, and Tegawendé F Bissyandé. 2023. The Best of Both Worlds: Combining Learned Embeddings with Engineered Features for Accurate Prediction of Correct Patches. *ACM Transactions on Software Engineering and Methodology* (2023).

[52] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant–How far is it? *arXiv preprint arXiv:2304.11938* (2023).

[53] Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and TegawendÉ F BissyandÉ. 2022. Is this Change the Answer to that Problem? Correlating Descriptions of Bug and Code Changes for Evaluating Patch Correctness. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[54] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. 2022. What makes a good commit message?. In *Proceedings of the 44th International Conference on Software Engineering*. 2389–2401.

[55] 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.

[56] 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. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

[57] 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.

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[59] 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.

[60] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. 2016. A survey of transfer learning. *Journal of Big data* 3, 1 (2016), 1–40.

[61] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*. 38–45.

[62] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[63] Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. 2019. Commit message generation for source code changes. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. 3975–3981.

[64] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering*. 1506–1518.

[65] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696* (2017).

[66] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2019. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system. *Empirical Software Engineering* 24 (2019), 33–67.

[67] Shufan Zhou, Beijun Shen, and Hao Zhong. 2019. Lancer: Your code tell me what you need. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1202–1205.

[68] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-Agnostic Representation Learning of Source Code from Structure and Context. In *International Conference on Learning Representations (ICLR 2019)*.