Divide-and-Conquer: Automating Code Revisions via Localization-and-Revision

SHANGWEN WANG, National University of Defense Technology, China
BO LIN[*], National University of Defense Technology, China
LIQIAN CHEN, National University of Defense Technology, China
XIAOGUANG MAO, National University of Defense Technology, China

Despite its effectiveness in ensuring software quality, code review remains a labor-intensive and time-consuming task. In order to alleviate this burden on developers, researchers have proposed the automation of code review activities, particularly focusing on automating code revisions. This automation can benefit both code authors, as they are relieved from the manual task of code revision, and code reviewers, as they are spared from addressing minor code flaws through manual comments. While current code revision approaches have shown promising results, they typically operate within a single phase, in which the code requiring revision is treated as the input of a deep learning model, and the revised code is directly generated through a sequence-to-sequence transformation. Consequently, these approaches tackle both the challenges of localization (i.e., where to revise) and revision (i.e., how to revise) simultaneously. Attempting to handle the entire complex process with a single model goes against the principle of "Divide-and-Conquer", which encourages breaking down complex problems into smaller sub-problems and addressing them individually. In fact, we have observed that existing code revision approaches often yield inaccurate results in both the localization and revision phases. In this paper, we present a two-phase code revision approach that aims to overcome the aforementioned limitations by adhering to the "Divide-and-Conquer" principle. Our approach comprises two key components: a localizer, responsible for identifying the specific parts of the input code that require revisions, and a reviser, tasked with generating the revised code based on the localization result. Extensive experiments conducted on two widely-used datasets demonstrate the substantial superiority of our approach over existing code revision approaches. For instance, when revising code based on the code reviewer's comments, our approach achieves a success rate of over 20% in implementing the ground-truth code revisions. In comparison, the widely-used pre-trained model CodeT5 achieves a success rate of less than 16% on the same test set, which contains 16K+ cases.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**; **Maintaining software**; **Automatic programming**.

Additional Key Words and Phrases: Code Review, Code Revision, Localization.

[*]Bo Lin is the corresponding author.

Authors' addresses: Shangwen Wang, wangshangwen13@nudt.edu.cn, Key Laboratory of Software Engineering for Complex Systems, College of Computer, National University of Defense Technology, Changsha, China; Bo Lin, linbo19@nudt.edu.cn, Key Laboratory of Software Engineering for Complex Systems, College of Computer, National University of Defense Technology, Changsha, China; Liqian Chen, lqchen@nudt.edu.cn, Key Laboratory of Software Engineering for Complex Systems, College of Computer, National University of Defense Technology, Changsha, China; Xiaoguang Mao, xgmao@nudt.edu.cn, Key Laboratory of Software Engineering for Complex Systems, College of Computer, National University of Defense Technology, Changsha, China.

1049-331X/2024/9-ART $15.00
https://doi.org/10.1145/xxxxxxx

ACM Reference Format:
Shangwen Wang, Bo Lin, Liqian Chen, and Xiaoguang Mao. 2024. Divide-and-Conquer: Automating Code Revisions via Localization-and-Revision. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (September 2024), 25 pages. https://doi.org/10.1145/xxxxxxx

1 INTRODUCTION

Code review is a critical activity in modern software development [1, 35, 36], during which the potential bugs are prevented and the software quality is thus ensured [2, 34]. Given the easily-recognized benefits, code reviews are widely adopted in both open-source projects [44] and industrial software [45]. However, these advantages often require significant human efforts, as code reviewers have to manually inspect the code and code authors have to manually make revisions to address both functional (e.g., compilation and testing) and non-functional (e.g., readability and maintainability) concerns that have been identified. For instance, it was reported that Microsoft Bing requires more than 3K reviews per month [43]. Quantitatively, developers spend an average of more than six hours per week reviewing code submitted by others [3]. Moreover, reviewing forces developers to switch context away from their current work, which may also increase the efforts spent in the code review process [6].

To reduce the burden on developers, researchers have proposed a number of approaches to automate code review activities [49, 53, 54]. In this domain, there is a particular emphasis on automating code revisions. This can involve proactively addressing potential flaws before submitting the code (i.e., Code Revision before Review) [49, 53] or making intended code changes in response to comments from code reviewers (i.e., Code Revision after Review) [27, 67], both of which are considered as effective to boost the code review process. For instance, Tufano *et al.* made the first step in this direction by using a supervised Transformer model [54], and they later applied self-supervised pre-training techniques on such tasks [53]. We will refer to the existing studies as *code revision approaches* hereafter.

Despite some promising results achieved, existing code revision approaches do not follow a common practice of making code changes. Specifically, the process of making a code change typically consists of two primary steps: localization (addressing the question of *where to change*) and revision (addressing the question of *how to change*). For instance, the widely-studied automatic program repair (APR) techniques usually contain a fault localizer, which identifies the buggy statements, and a patch generator, which performs program transformations to generate candidate patches [13, 32]. In contrast, all the existing code revision approaches perform the task within a single phase in which a deep learning model is utilized to change the submitted code, probably with the review comment, into the code revised by the developer through a sequence-to-sequence transformation. In this paradigm, the model is required to perform both the localization and revision tasks simultaneously, which means that it needs to first understand which parts of the code require modifications and then autonomously make the necessary revisions to such code. Having an individual model fulfill such a complex process contradicts a foundational principle in software engineering, i.e., the "Divide-and-Conquer" principle, where it is encouraged to break down a complex problem into smaller, more manageable sub-problems for facilitating problem-solving [21]. Indeed, we have observed (and will illustrate with specific examples in Section 3) that models operating within this paradigm could either pinpoint inaccurate locations for making modifications or generate incorrect modifications, which can be considered as two primary limitations of existing code revision approaches. Previous studies in the software engineering domain have shown that splitting the original problem into several sub-problems and solving them independently would result in effectiveness enhancement [23, 29, 38]. We therefore postulate that the effectiveness of code revision approaches could also be boosted if they could follow the "Divide-and-Conquer" principle.

Based on the above idea, in this paper, we propose a two-phase approach named `CodeReviser` for automating code revisions. `CodeReviser` consists of two main components, i.e., a localizer to identify which parts of the code need to be revised, and a reviser to perform the modifications based on the localization results of the localizer. The behind intuition is that by breaking down the code revision task into two comparatively trivial sub-tasks, the localizer and reviser could effectively accomplish their respective roles, thereby overcoming the aforementioned two limitations of existing code revision approaches, and combining these two components would lead to more qualified code revision results at the end. To provide qualified localization results, the localizer of `CodeReviser` mainly faces two challenges, i.e., how to make use of the domain knowledge encoded in the pre-trained code models and how to localize multiple code entities that need to be changed. To tackle these challenges, the localizer in `CodeReviser` employs a novel *generative localization* strategy, which offers a dual rationale. First, by formulating the localization task as a sequence-to-sequence transformation, the localizer can leverage an encoder-decoder architecture and initialize its parameters using pre-trained code models such as CodeT5 [62], which enables it to incorporate domain knowledge about programming languages captured from vast amounts of data. Second, by generating the localization result rather than selecting a fixed number of entities, the localizer can theoretically provide a more flexible outcome, allowing for the localization of an arbitrary number of code entities. Building on the impressive code generation capabilities demonstrated by pre-trained code models [5, 57, 58, 62, 66], the reviser in `CodeReviser` leverages the existing CodeT5 model and undergoes additional fine-tuning specific to its task. Consequently, from the holistic view, both the localizer and the reviser components of `CodeReviser` adhere to the sequence-to-sequence paradigm, where the former translates the input code into an intermediate representation that includes the localization result; while the latter takes this intermediate representation as input and generates the revision.

To assess the effectiveness of `CodeReviser`, we perform extensive experiments on two widely-used datasets in the code revision domain, i.e., $\textit{Trans-Review}_{data}$ [54] and $\textit{T5-Review}_{data}$ [53]. Results reveal that `CodeReviser` outperforms the state-of-the-art code revision approaches to a large extent on both the Code Revision before Review and the Code Revision after Review tasks. For instance, in terms of the Code Revision after Review task, `CodeReviser` achieves Exact Match values (indicating how often an approach generates a code revision that is identical to that made by developers) of 40.8% and 20.5% on the two datasets, respectively, exceeding the best-performing baseline, i.e., CCT5 [28], by 17% and 22%. Moreover, a thorough analysis reveals that `CodeReviser` exhibits more superior proficiency than state-of-the-art approaches in both precisely identifying modification locations and generating accurate revisions. These findings suggest that adhering to the "Divide-and-Conquer" principle holds promise in mitigating the limitations of current code revision approaches.

In summary, our study makes the following major contributions:

- **Significance:** Through case analysis, we identify the two primary challenges of current code revision approaches as: ① pinpointing the accurate locations for making code revisions, and ② generating the correct modifications.
- **Approach:** We propose `CodeReviser`, a two-phase code revision approach that adheres to the "Divide-and-Conquer" principle. It employs a localizer to identify the specific parts of the code requiring revision, and a reviser to generate the necessary code changes. In particular, the localizer employs a novel *generative localization* strategy, which empowers the localizer to leverage the

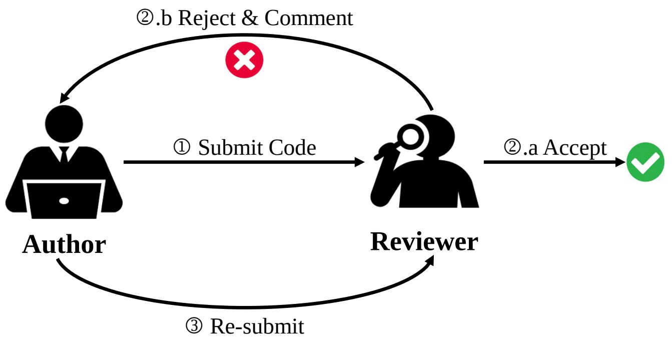

Fig. 1. A brief illustration of the code review process.

capabilities of pre-trained code models while also enables it to identify multiple code segments in need of revision.

- **Experiments:** We conduct extensive experiments in four different settings (two datasets × two code revision tasks), and find that our approach consistently outperforms existing approaches. We also open source the artifacts of this study at **https://zenodo.org/record/8373320** to facilitate replications and follow-up studies.

2 BACKGROUND

2.1 Code Review Process and the Involved Code Revision Tasks

Figure 1 illustrates the process of code reviews. Typically, there are three main activities in this process. In the first step, code authors submit the code for review. Then, code reviewers check the submitted code and judge its correctness. The result of this step would be either accepting the code and merging it into the main repository or rejecting the code and explaining how to improve the code with a detailed comment. Later, the code authors revise the code, address the received comment, and resubmit the code for review again. Note that the resubmitted code may receive an additional comment made by the reviewer, which could result in a second revision. Such a process ends when the reviewer accepts the revised code.

During this process, there are mainly two derived code revision tasks that can be automated to enhance software development. We follow the existing studies to formulate these two tasks [49, 53, 54, 68].

Code Revision before Review (CRB). This type of revision could happen before the code authors submit their code (i.e., before Step ①). The benefits of this type of revision are twofold: on the one hand, it could help the authors address some simple flaws and thus improve the quality of the code before the code review; on the other hand, it also releases the burden of the reviewers on commenting on simple flaws during the review. Formally, given a code snippet to submit for review C_s, and its revised version C_r that fulfills the recommended code changes from reviewers, this task is formulated as: $f(C_s) \rightarrow C_r$, where f is a transformation approach, e.g., a deep learning model.

Code Revision after Review (CRA). This type of revision aims at revising the code based on reviewer's comments (i.e., happens between Step ②.b and Step ③). The benefits of this type of revision are also twofold: on the one hand, the reviewer is able to attach to the comment a preview of how the code could look like by addressing his/her concerns; on the other hand, the author is able to revise the code more quickly since the initial revised version can be automatically generated. Formally, given C_s and C_r together with a comment R_{nl} written by the reviewer in natural language, this task can be formulated as: $f(C_s, R_{nl}) \rightarrow C_r$, where f is another transformation approach.

2.2 Existing Code Revision Approaches

In recent years, researchers have proposed a number of approaches based on the popular Transformer architecture [55] to automate code revisions. In the following, we briefly introduce the existing approaches in the literature.

Trans-Review [54] applies the Transformer architecture on the code revision tasks through a sequence-to-sequence (seq2seq) translation paradigm (i.e., translating code before review into code after review), and such a paradigm is adopted by all the follow-up studies going to be introduced below. The main point of this approach is that to reduce the vocabulary size, the literals and identifiers in the code are abstracted, e.g., the method name `hasProperties` will be assigned the ID of `METHOD_1`.

AutoTransform [49] utilizes a Byte-Pair Encoding (BPE) tokenization [46] to overcome the limitation of Trans-Review on generating new identifiers. Specifically, although Trans-Review could be able to predict the correct ID for a new identifier or literal in the revised code (e.g., `VAR_10`), it cannot map the ID back to the actual content (e.g., `sidePanel`). BPE splits a code token into a sequence of frequently occurring sub-tokens, reducing the vocabulary size while preserving the ability of generating new tokens.

T5-Review [53] leverages the Text-To-Text Transfer Transformer (T5) architecture [40]. The model is pre-trained on a dataset collected from Stack Overflow and the CodeSearchNet dataset [18], with the Masked Language Modeling (MLM) pre-training task where the model is asked to predict some randomly masked contents.

CodeT5 [62] is another pre-trained model. Beyond the MLM task, it is also trained to tag identifiers in the code and further predict the detailed identifier names. The recent reproduction study [68] shows that CodeT5 outperforms the aforementioned approaches on code revision tasks.

NatGen [5], an extension of CodeT5, introduces an additional pre-training task where the model is trained to transform unnatural code into a developer's original writing style. This pre-training process aims to strengthen the model's grasp of code semantics, enabling it to produce code that closely resembles human-written code. The ultimate result could be that its proficiency in code revision tasks is enhanced.

CoditT5 [67] builds on top of CodeT5 and designs a new pre-training task to enable the model to reason about edits. Specifically, before outputting the revised code, the model is trained to generate an edit plan first, which entails the detailed operations that are going to be performed on the given code. It is expected that the edit plan can guide more accurate and precise code revisions.

CodeReviewer [27] also initializes the model parameters from CodeT5 and continuously pre-trains the model on a dataset collected from code review processes (i.e., consisting of code diffs and review comments). This model is expected to be more capable of understanding code changes since during the pre-training, the input to the model is in the code diff format.

CCT5 [28] also takes code diffs as inputs in the pre-training phase. The difference between this model and CodeReviewer mainly lies in its pre-training on a dataset gathered from software evolution processes (i.e., consisting of code commits and commit messages). Moreover, it also acquires the knowledge about program structure during the pre-training by predicting the code diff based on the data flow change.

Summary. From the above introduction, it is evident that although existing code revision approaches may vary in their chosen pre-training tasks, they generally operate within a seq2seq paradigm in which the submitted code is translated into its revised version through an individual deep learning model. This necessitates the model to not only identify errors in the code but also autonomously generate revisions for those errors. As we will demonstrate in the next section,

```
// Oracle Code Revision
public HueBridgeNupnpDiscovery() {
-    super(SUPPORTED_THING_TYPES, DISCOVERY_TIMEOUT);
+    super(SUPPORTED_THING_TYPES, DISCOVERY_TIMEOUT, false);
}

// Code Revision by CodeT5
public HueBridgeNupnpDiscovery() {
-    super(SUPPORTED_THING_TYPES, DISCOVERY_TIMEOUT);
+    super(false, DISCOVERY_TIMEOUT);
}
```

Listing 1. The oracle code revision and the revision generated by CodeT5 for the method `HueBridgeNupnpDiscovery`.

attempting to have a single model perform such a complex process may lead to its ineffectiveness in both steps, which could be considered as the limitations of current code revision approaches.

3 MOTIVATING EXAMPLES

In this section, we demonstrate two cases where CodeT5 [62], the state-of-the-art code revision approach [68], fails to generate the desired revision. We fine-tune and test the CodeT5-base model on the widely-used $T5\text{-}Review_{data}$ dataset [53], and both demonstrated cases are from the test set of this dataset.

In the first case shown in Listing 1, the code reviewer leaves a comment "Call Super with false as a parameter", which indicates that `false` should be used as a parameter in the function call. However, in the original code, the function call only passes two constants as arguments (i.e., `SUPPORTED_THING_TYPES` and `DISCOVERY_TIMEOUT`). To perform this revision, the developer inserts `false` as the third argument. In contrast, CodeT5 replaces the first argument with `false`, which leads to an incorrect function call. This indicates that CodeT5 comprehends the requirement from the code reviewer is to use `false` as a function argument, but it unfortunately locates a wrong place to pass the `false` parameter. From the coding conventions of Java language,[1] parameters with the boolean type (e.g., `true` and `false`) are typically placed at the end of the parameter list. This example demonstrates that when the submitted code is transformed directly into the revised code, the code revision model may struggle to identify the specific locations for modifications, resulting in the generation of inaccurate revisions.

In the second case shown in Listing 2, the code reviewer asks for a code change by commenting "Use a constant for "://"", indicating that the string `"://"` in the code snippet should be replaced by a constant with a descriptive name, which could potentially enhance the readability of the code compared with the original hard-coded value [7]. To implement the necessary modification, the developer utilizes a constant `PROTOCOL_SEPARATOR`, which aptly represents the semantic information of the string, to serve as a replacement. Indeed, the string `"://"` is usually used to separate a Uniform Resource Locator (URL). In detail, the URL generally follows the structure of "<protocol>://<hostname>:<port>/", in which the string `"://"` separates the protocol used to access the resource and the hostname that identifies the server. However, CodeT5 uses another constant `URL_PROTOCAL` to replace the string, which incorrectly represents its meaning. Specifically, the protocol of the URL typically refers to HTTP, HTTPS, FTP, etc., but it is unrelated to the replaced string. As a result, CodeT5 generates an inaccurate replacement. This example demonstrates that when the submitted code is transformed directly into the revised code, the code revision model

[1] https://www.baeldung.com/java-best-practices-many-parameters-method

```
// Oracle Code Revision
private void setPublicURIs(final FileSystem fileSystem) {
    final String[] uris = fileSystem.toString().split("\\\\r?\\\\n");
    final List <PublicURI> publicURIs = new ArrayList <> (uris.length);
    for (final String s: uris) {
-        final int protocolStart = s.indexOf("://");
+        final int protocolStart = s.indexOf(PROTOCOL_SEPARATOR);
         publicURIs.add(getPublicURI(s, protocolStart));
    }
    repo.setPublicURIs(publicURIs);
}

// Code Revision by CodeT5
private void setPublicURIs(final FileSystem fileSystem) {
    final String[] uris = fileSystem.toString().split("\\\\r?\\\\n");
    final List <PublicURI> publicURIs = new ArrayList <> (uris.length);
    for (final String s: uris) {
-        final int protocolStart = s.indexOf("://");
+        final int protocolStart = s.indexOf(URL_PROTOCOL);
         publicURIs.add(getPublicURI(s, protocolStart));
    }
    repo.setPublicURIs(publicURIs);
}
```

Listing 2. The oracle code revision and the revision generated by CodeT5 for the method `setPublicURIs`.

may struggle to generate accurate modifications even if it identifies which parts of the code should be revised.

These two cases demonstrate that performing the code revision tasks in a seq2seq transformation paradigm mainly faces two limitations, i.e., ① the inaccurate identification of the specific locations for modifications and ② the inability of accurately generating desired modifications at the ground-truth locations (the locations where the developers make revisions). Drawing on such observations, we hypothesize that the effectiveness of code revision approaches could be improved by either enhancing the accuracy of localization results or enhancing their ability to generate accurate modifications. Inspired by the previous studies [25, 26, 29, 38] which split a complex task into several sub-tasks, effectively address each sub-task, and successfully achieve effectiveness enhancements for the original complex task in the end, we also propose to adopt the "Divide-and-Conquer" principle to address the aforementioned limitations. Specifically, we can design a two-phase approach in which a localizer first identifies the localizations for modifications, followed by a reviser that makes the specific revisions. The underlying assumption here is that, by doing so, both the localizer and reviser handle relatively trivial tasks compared to the direct transformation of the input code into the revised code, which allows them to effectively address their respective assignments. Ultimately, by combining these two models, we can enhance the effectiveness of code revisions. Indeed, experiment results show that our approach can generate oracle outputs (i.e., the developer-provided code) in the two cases shown in Listing 1 and Listing 2.

4 METHODOLOGY

The overall workflow of `CodeReviser` is shown in Figure 2. Following the "Divide-and-Conquer" principle, `CodeReviser` consists of a localizer, which identifies which parts of the code need to be changed, and a reviser, which revises the code based on the output of the localizer. We introduce the details of the two components in the following.

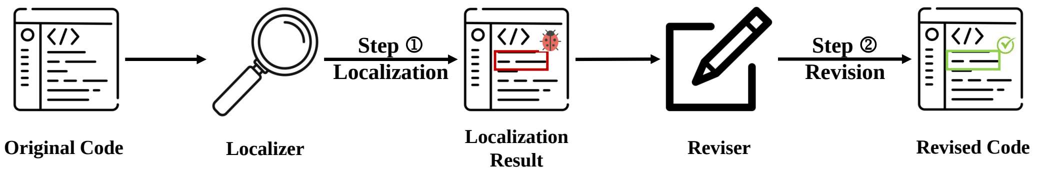

Fig. 2. The workflow of CodeReviser.

4.1 Localizer

Generative localization. Given a code snippet, the target of the localizer is to identify the code tokens that need to be changed. We propose to identify the token or token spans that need to be changed since as observed in cases like Listings 1 and 2, code revisions typically pertain to specific parts of a statement, such as an expression or an individual code token. Therefore, the traditional statement-level localization in the APR domain [69] would provide too coarse-grained localization results for code revision approaches. In the literature, there is an approach named BEEP [59] that can take the buggy method as the input and predict which code token is incorrect. However, directly applying this approach on our task is inappropriate due to two limitations. First, the BEEP model is trained from scratch through a supervised learning manner. However, as demonstrated by recent studies [39, 57, 66, 68], pre-trained code models have outperformed traditional supervised deep learning models on a wide range of software engineering tasks, potentially due to the domain knowledge of programming languages captured during the pre-training. As a result, the exclusion of utilizing pre-trained code models leads to a missed opportunity to leverage the extensive domain knowledge encoded within them. Second, the BEEP model (with the help of the pointer generator network [56]) assigns a value to each input token, which indicates the probability of it being associated with a bug, and always selects the one with the highest probability as the buggy token. However, code revisions usually involve multiple code tokens. For instance, as revealed by the existing study [48], over half of the patches in the Defects4J benchmark [20], which contains hundreds of bug-fixing patches mined from projects on GitHub, involve modifications in more than four statements. Consequently, it is hard for BEEP to provide an accurate localization result when multiple code tokens need to be changed.

To overcome the aforementioned limitations, the localizer of CodeReviser adopts a *generative localization* strategy that follows the seq2seq paradigm and translates the input code token sequence into another sequence, where the localization result is included. The rationale of such a design is twofold. On one hand, by treating the localization task in a seq2seq paradigm, we can borrow the weapon of pre-trained code models with encoder-decoder architectures such as CodeT5, equip the localizer with commonsense knowledge upon initialization instead of random initialization, and therefore overcome the first limitation. On the other hand, the localization result is generated rather than selected, so that such a strategy can theoretically localize an arbitrary number of code tokens as long as the model generates accurate results, which addresses the second limitation.

Representation of the input. Formally, suppose the token sequence of the submitted code is $T = t_1, t_2, \dots, t_n$ with the length of n, and the token sequence of the reviewer's comment is $C = w_1, w_2, \dots, w_x$ with the length of x. For the CRB task, the input to the localizer I_{loc} is the code token sequence: $I_{loc} = T$; while for the CRA task, the input to the localizer is the concatenation of T and C: $I_{loc} = w_1, w_2, \dots, w_x, [SEP], t_1, t_2, \dots, t_n$, where $[SEP]$ is a special token indicating the separation between the code and the comment.

Representation of the output. We identify several (m) code token spans that are modified by developers through comparing the original code token sequence with the sequence of the oracle code, i.e., the code revised by the developer (by performing token-level code diffs with the help

```
/* Case 1 */
// Input code token sequence
... super ( SUPPORTED_THING_TYPES , DISCOVERY_TIMEOUT ) ...
// Oracle localization result
... super ( SUPPORTED_THING_TYPES , DISCOVERY_TIMEOUT [START] [END] ) ...

/* Case 2 */
// Input code token sequence
private void ... protocolStart = s . indexOf ( "://" ) ; ...
// Oracle localization result
private void ... protocolStart = s . indexOf ( [START] "://" [END] ) ; ...
```

Listing 3. The oracle localization results for the two motivating examples.

of the *difflib*[2] library of Python). Such m code token spans are denoted as $TS = \{ts_1, ts_2, \ldots, ts_m\}$ where $ts_i = t_{i_{start}}, \ldots, t_{i_{end}}$ is a consecutive token sequence from T with the length of $|i_{end} - i_{start} + 1|$, $1 \leq i_{start} \leq i_{end} \leq n$. It should be noted that a token span can represent various entities in the original code, such as a changed code token, a modified expression, or even a sequence of consecutive statements. If $i_{start} = i_{end}$, it means that compared with the original code, the oracle code inserts some contents starting from the position $t_{i_{start}}$. The identified token spans correspond to the specific sections in the original code that require modifications. The objective of the localizer is to identify these spans, thereby facilitating the subsequent task of the reviser. To that end, we use two special tokens (i.e., $[START]$ and $[END]$) to signify the beginning and ending positions of each modified token span and train the localizer to accurately predict the positions of these two special tokens in its output. As a result, the oracle output of the localizer is denoted as $T_{loc} = t_1, t_2, \ldots, [START], t_{1_{start}}, \ldots, t_{1_{end}}, [END], \ldots, [START], t_{m_{start}}, \ldots, t_{m_{end}}, [END], \ldots, t_n$. In Listing 3, we demonstrate the oracle localization results for the two motivating examples in Section 3. The first case is a vivid insertion example whose i_{start} equals to i_{end}. In such a case, the localizer identifies the insertion point and downstream reviser is expected to insert two tokens at that point, i.e., `,` and `false`. The second case has one modified code token span in the original code that represents an individual token (the string `"://"`). The oracle output of the localizer should start the token span with a $[START]$ token and end it with an $[END]$ token (as highlighted in yellow).

It is widely observed that the outputs of large code models can be unstable. There is thus a concern that whether we can ensure the generated localization result is in the required form (i.e., inserting a number of [START] and [END] token pairs into the original code token sequence). To answer this question, we have checked our experiment results and confirmed that the outputs of the localizer are in the ideal form. One possible explanation would be that during fine-tuning, the model has learned the domain knowledge about how to perform such a task.

Loss calculation. Formally, the loss during the training process can be described as:

$$\mathcal{L}(\theta) = \sum_{j=1}^{L_{loc}} -logP_\theta(t_j | I_{loc}, t_{<j}) \tag{1}$$

where I_{loc} is the input of the model whose contents depends on the ongoing task, θ is the set of parameters of the localizer, L_{loc} denotes the number of tokens in the oracle output T_{loc} (mathematically, $L_{loc} = n + 2 * m$), and $t_{<j}$ is the token sequence generated so far.

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

4.2 Reviser

Given the output token sequence of the localizer, the target of the reviser is quite straightforward, that is, to generate the revised code. Considering the superior generation capability of the pre-trained code models [5, 28, 62], the reviser of `CodeReviser` also follows the encoder-decoder architecture to seek for a promising effectiveness.

During training, for the CRB task, the input to the reviser I_{rev} is the oracle localization result T_{loc}; while for the CRA task, the input I_{rev} is the concatenation of T_{loc} and C (the behind intuition is that the reviewer's comment could also help guide the revision process). Formally, suppose the token sequence of the oracle code associated with T is $T_{ora} = t'_1, t'_2, \ldots, t'_{n'}$ where n' denotes the length of the token sequence after revision, the loss during the training process can be described as:

$$\mathcal{L}(\beta) = \sum_{j=1}^{n'} -logP_{\beta}(t'_j | I_{rev}, t'_{<j}) \tag{2}$$

where I_{rev} is the input of the reviser, β is the set of parameters of the reviser, and $t'_{<j}$ is the token sequence generated so far.

It should be noted that during testing, the input to the reviser is not the oracle localization result T_{loc}, but the output obtained from the localizer.

4.3 Implementation Details

Our approach is implemented with the popular deep learning development framework PyTorch.[3] All the experiments are performed on a server with 4 NVIDIA GeForce RTX 4090 GPUs. We adopt the same architecture as the T5 model [40] for both the localizer and the reviser. Specifically, each model is based on the encoder-decoder architecture with 12 Transformer encoder layers and 12 Transformer decoder layers. In each layer, there are 12 attention heads and the hidden size is set to 768.

Both the localizer and the reviser are initialized with the parameters released by CodeT5. This is because the CodeT5 model captures some commonsense knowledge about programming languages and natural languages, and has served as the starting point for a number of recent studies [5, 27, 28, 67]. After that, these two models are further fine-tuned on the code revision datasets. During fine-tuning, the learning rate and batch size for the two models are set to 5e-5 and 32, following the existing study [28].

5 EXPERIMENTAL DESIGN

5.1 Research Questions

We have conducted several experiments to evaluate our approach `CodeReviser`. Specifically, we seek to answer the following research questions:

RQ1. Effectiveness of Code Revision before Review. What is the effectiveness of `CodeReviser` on the CRB task? To answer this question, we compare the performance of `CodeReviser` with those of the state-of-the-art approaches on the CRB task.

RQ2. Effectiveness of Code Revision after Review. What is the effectiveness of `CodeReviser` on the CRA task? To answer this question, we compare the performance of `CodeReviser` with those of the state-of-the-art approaches on the CRA task.

RQ3. Localization Ability. To what extent can `CodeReviser` generate revisions at the correct locations? To answer this question, we check the output of `CodeReviser` and investigate the

[3] https://pytorch.org/

Table 1. The statistics of our evaluation datasets.

Dataset	Training	Validation	Test
Trans-Review$_{data}$	13,756	1,719	1,719
T5-Review$_{data}$	134,239	16,780	16,780

proportion of the cases where `CodeReviser` performs modifications at the ground-truth locations (i.e., the locations where the developers make revisions).

RQ4. Revision Ability. To what extent can `CodeReviser` generate accurate modifications at the ground-truth locations? To answer this question, we investigate how often `CodeReviser` generates correct revisions under the condition that it identifies the ground-truth locations.

To answer RQ3 and RQ4, we conduct post-processing after obtaining the revision results by analyzing (1) whether the approach revised the code at the correct locations and (2) how well the approach revises the code if it knows the ground-truth location. It should be noted that the inputs to the models in RQ3 and RQ4 are identical to those in the first two RQs (i.e., we do not provide the ground-truth locations to any approach in RQ3 or RQ4).

5.2 Datasets

Following existing studies [67, 68], we include two widely-used datasets for evaluating `CodeReviser`, namely *Trans-Review*$_{data}$ [54] and *T5-Review*$_{data}$ [53]. These two datasets contain the code change information as well as the reviewers' comments, and thus can be used for both the CRB task and the CRA task. As a result, both datasets are utilized to address all the four research questions. When performing the CRB task (e.g., RQ1 and the CRB perspectives in RQ3 and RQ4), only the original code is used as the input to the model; while for the CRA task (e.g., RQ2 and the CRA perspectives in RQ3 and RQ4), both the original code and reviewers' comments are used as the input. Note that all the code changes in the datasets are from the method-level granularity, which is the most widely-studied granularity in the code revision domain [51, 52]. The detailed statistics of the datasets are shown in Table 1.

Trans-Review$_{data}$ is collected from 6,388 projects in Gerrit and 2,566 projects from GitHub. Tufano *et al.* manually create a set of heuristics to filter out noisy data. For instance, a comment with only one word or requests to add some test code is removed. They also require that the submitted code and the revised code should each have a maximum of 100 tokens, and the revised code must not introduce new tokens that are not originally present in the submitted code, which may simplify the task for the model.

T5-Review$_{data}$ is mainly mined from 4,901 Java open-source projects from GitHub, which have at least 50 pull requests, 10 contributors, and 10 stars. Tufano *et al.* also preprocess the collected data by various means, e.g., by excluding the non-English comments, removing any emoji and non-ascii characters from the comments, and preforming the deduplication.

5.3 Baselines

We have introduced existing code revision approaches in Section 2.2. In this work, we choose to use **CodeT5**, **NatGen**, **CoditT5**, **CodeReviewer**, and **CCT5** as the baselines. According to a recent study [68], CodeT5 outperforms other popular pre-trained code models like CodeBERT [11] and GraphCodeBERT [16], as well as previous code revision approaches including Trans-Review, AutoTransform, and T5-Review, making it the top performer in code revision tasks. The other four selected baselines have all strengthened the capability of CodeT5 in different aspects. In particular, NatGen is trained to transform unnatural code into the form originally written by developers, potentially enabling it to better revise code; CoditT5 is trained to generate edit plans, allowing it to reason about code edits; whereas CodeReviewer and CCT5 are trained using code change data, making them more proficient in comprehending code modifications. As a result, the selected baselines can represent the state of the art in the code revision domain well.

5.4 Metrics

To answer RQ1 and RQ2, we report the performance of each code revision approach by comparing the revised code with the oracle code (i.e., the code revised by the developer) from three different aspects.

- **Exact Match (EM)**, also known as *perfect prediction*, focuses on assessing whether the code revised by a model exactly matches the oracle. Specifically, if the token sequence of the model's output is identical to that of the ground truth, then EM = 1, otherwise, EM = 0. This is a strict metric, as even a subtle difference from the oracle can result in the model's output receiving a score of 0. This metric has been widely adopted in the evaluation of previous studies [49, 53, 54].
- **CodeBLEU** [42] assesses the similarity between the generated code and the oracle code. The central concept behind this metric lies in recognizing that, unlike plain text, code snippets inherently contain structural and semantic information in addition to lexical contents. Consequently, it assesses the generated code against the oracle code from various perspectives, encompassing token similarity (denoted as TM, calculated using standard BLEU metrics), syntactic similarity (denoted as SM, determined through AST structure comparisons), and dataflow similarity (referred to as DM, evaluated based on data flow analysis). The final evaluation score is a composite measure that integrates these similarities, offering a comprehensive assessment of the quality of the generated code (more details about this metric are referred to the original study [42]). This metric has been widely used to assess the quality of the generated code by previous studies [57, 66].
- **Edit Progress (EP)** [68] measures the progress made by the generated code on changing the submitted code into the oracle code. The basic idea of this metric is that in certain cases, the generated code might have reduced the number of errors in the submitted code, even if it is not an exact match to the ground truth. In such cases, the Exact Match metric falls short in fully capturing the effectiveness of code revision approaches. Let $\tilde{C}$, $\bar{C}$, and $\hat{C}$ denote the initial submitted code, the ground-truth code, and the revised code generated by the model, respectively. Then, the value of this metric represents the percentage of the modifications that are successfully accomplished by generating $\hat{C}$ from $\tilde{C}$, compared with the total modifications required to transform $\tilde{C}$ into $\bar{C}$. Formally, this metric can be calculated as:

$$Edit\ Progress = \frac{|D_{\tilde{C}\to\bar{C}}| - |D_{\hat{C}\to\bar{C}}|}{|D_{\tilde{C}\to\bar{C}}|} \tag{3}$$

  where $|D_{A\to B}|$ denotes the token-level edit distance [24] between a token sequence A and a token sequence B. In this paper, we calculate and report the average EP value on all the cases from the test set. This value indicates to what extent a code revision approach can release developers' burdens from manual revisions: the higher, the better. It should be noted that an exact match would achieve a 100% EP score while a model's output $\hat{C}$ could also achieve a negative EP value if it generates more errors.

To answer RQ3, we propose a metric named **Localization Accuracy (LA)** to measure the localization ability of different code revision approaches. The basic idea of this metric is to assess if an approach can accurately identify which parts of the code should be revised. We recall that in Section 4.1, we have introduced that we can identify m code token spans (denoted as TS) from the token sequence of the submitted code that are modified by the developer. Such a result is obtained by performing token-level code diffs between the submitted code and the oracle code. To calculate the value of this metric, we also identify code token spans (denoted as $TS^{'}$) from the submitted code that are modified by the code revision approach (by comparing the original code token sequence with the sequence of the generated code). An approach is deemed successful in identifying the code requiring revisions if $TS^{'}$ exactly matches TS. Take Listing 2 as an example. By performing token-level code diffs between the submitted code and the oracle code, we note the TS contains only one token span, which involves a single code token `"://"`. Then, by performing token-level code diffs between the submitted code and the generated code, we note the $TS^{'}$ also merely contains such a single token. This implies that in such a case, the code revision approach performs revisions at the same locations where developers make code revisions, accurately identifying which parts of the code should be modified. Finally, the value of LA is the percentage of the submitted code whose $TS^{'}$ is identical to TS. Formally, suppose the total number of samples in the test set is N, the number of the submitted code whose $TS^{'}$ is identical to TS is num_1, then LA can be calculated as:

$$Localization\ Accuracy = P(TS = TS^{'}) = \frac{num_1}{N} \tag{4}$$

To answer RQ4, we propose a metric named **Revision Accuracy (RA)** to measure the revision ability of different code revision approaches. The basic idea of this metric is to check if an approach can generate correct revisions if it already knows the ground truth locations. Consequently, this metric is essentially a conditional probability. Formally, suppose the number of the submitted code whose $TS^{'}$ is identical to TS is num_1, the number of perfect predictions (the generated code $\hat{C}$ and the ground-truth code $\bar{C}$ are the same) is num_2, then RA can be calculated as:

$$Revision\ Accuracy = P(\hat{C} = \bar{C} | TS = TS^{'}) = \frac{num_2}{num_1} \tag{5}$$

5.5 Experiment Details

To mitigate the performance variance brought by different machines [66], we fine-tune and test all the baselines on our own server. To avoid potential implementation biases, in our experiments, we use the replication packages of CodeT5,[4] NatGen,[5] CoditT5,[6] CodeReviewer,[7] and CCT5.[8]

During fine-tuning, the learning rate and the batch size of the baselines are set according to their original studies. The best-performing model checkpoint on the validation set is used for testing. We feed each submitted code from the test set (and the reviewer's comment for the CRA task) into the approaches and obtain the generated code. Then we calculate the metrics by comparing the generated code with the ground truth.

6 EXPERIMENT RESULTS

6.1 RQ1: Effectiveness on the CRB Task

The results of different approaches on the CRB task are shown in Table 2. We observe that `CodeReviser` consistently outperforms the baselines with respect to all the three metrics. Specifically, on the $Trans\text{-}Review_{data}$ dataset, `CodeReviser` achieves an Exact Match of 19.3%, exceeding the best-performing baseline (CCT5 with 13.5%) by around 43%. The Edit Progree of `CodeReviser` reaches 48.4%, indicating that it is capable of addressing nearly half of the revisions needed to transform the submitted code into the oracle version. It is worth noting that the CodeBLEU values of different approaches exhibit minimal variation, ranging from 71.6% to 73.4%. This similarity can

[4]https://github.com/salesforce/CodeT5
[5]https://github.com/saikat107/NatGen
[6]https://github.com/EngineeringSoftware/CoditT5
[7]https://github.com/microsoft/CodeBERT/tree/master/CodeReviewer
[8]https://github.com/Ringbo/CCT5

Table 2. The effectiveness of different approaches on the code revision before review task (in %).

Dataset	Approach	Exact Match	CodeBLEU	Edit Progress
Trans-Review$_{data}$	CodeT5	8.7	72.9	44.6
	NatGen	10.7	71.6	43.5
	CoditT5	8.9	73.0	45.8
	CodeReviewer	12.4	73.0	46.0
	CCT5	13.5	72.6	45.0
	`CodeReviser`	**19.3**	**73.4**	**48.4**
T5-Review$_{data}$	CodeT5	4.7	73.8	7.6
	NatGen	4.5	73.4	7.2
	CoditT5	5.5	72.8	9.2
	CodeReviewer	5.6	72.4	9.3
	CCT5	5.7	73.2	7.2
	`CodeReviser`	**7.6**	**74.1**	**12.8**

be attributed to the fact that code revisions sometimes involve minor modifications (e.g., renaming a variable), resulting in a high degree of similarity between the original code and the revised code. This suggests that CodeBLEU may not be a differentiating metric when evaluating code revisions. However, it is important to highlight that the CodeBLEU value of `CodeReviser` remains higher than those of the baselines. This indicates that the code generated by `CodeReviser` exhibits the closest resemblance to the oracle code among all the approaches considered.

We observe the similar trends on the *T5-Review$_{data}$* dataset. Specifically, the Exact Match value of `CodeReviser` reaches 7.6%, which already exceeds that of the state-of-the-art CCT5 by 33% (7.6% vs. 5.7%). While the EM value of `CodeReviser` may appear small, it indicates that `CodeReviser` generates the oracle code for over 300 additional cases compared to CCT5 given the large size of this dataset. Similarly, `CodeReviser` achieves an Edit Progress of 12.8%, which is the only one that surpasses 10% among all the involved techniques. We conduct the Wilcoxon Signed-Rank Tests [63] to analyze the statistical significance of the difference between the effectiveness of `CodeReviser` and the baselines. We opt for non-parametric tests since they do not make assumptions about the distribution of the data or the equality of variance and thus provide a more flexible approach to analyzing experimental data [14, 29, 60]. Specifically, we compared the achieved Edit Progress values of `CodeReviser` on each individual test case against those achieved by the baselines (we exclude the CodeBLEU score in the analysis since it may not be a differentiating metric when evaluating code revisions, as observed from the experiment results). Such a comparison was conducted for the two datasets and the results show that `CodeReviser` significantly outperforms the baselines in both datasets. Specifically, all the p-values are less than 0.01 in the comparison results.

It is also worth highlighting that code revision approaches tend to achieve relatively higher performances on the *Trans-Review$_{data}$* dataset. As we have discussed in Section 5.2, this observation can be attributed to certain pre-processing steps employed on this dataset, such as limiting the number of tokens in the code. These pre-processing steps may reduce the complexity of code revisions, thus making it easier for the approaches to achieve better results on this particular dataset.

6.2 RQ2: Effectiveness on the CRA Task

The results of different approaches on the CRA task are shown in Table 3. We observe a phenomenon similar to that in Table 2, i.e., `CodeReviser` consistently outperforms the existing code revision approaches. In terms of the Exact Match metric, `CodeReviser` achieves impressive results on the *Trans-Review$_{data}$* and *T5-Review$_{data}$* datasets, with scores of 40.8% and 20.5%, respectively. These

Table 3. The effectiveness of different approaches on the code revision after review task (in %).

Dataset	Approach	Exact Match	CodeBLEU	Edit Progress
$Trans\text{-}Review_{data}$	CodeT5	30.2	77.2	59.8
	NatGen	32.8	78.6	59.5
	CoditT5	34.0	77.6	60.6
	CodeReviewer	32.9	78.5	61.2
	CCT5	34.9	79.0	61.8
	`CodeReviser`	**40.8**	**80.0**	**64.4**
$T5\text{-}Review_{data}$	CodeT5	15.9	75.9	25.3
	NatGen	16.6	76.3	23.8
	CoditT5	16.4	76.3	23.5
	CodeReviewer	16.6	76.0	26.3
	CCT5	16.8	76.1	25.7
	`CodeReviser`	**20.5**	**76.4**	**31.3**

scores surpass the best-performing baseline, i.e., CCT5, by 17% (40.8% vs. 34.9%) and 22% (20.5% vs. 16.8%). Regarding the CodeBLEU metric, `CodeReviser` continues to outperform the baselines, albeit with subtle improvements. However, it is worth noting that `CodeReviser` demonstrates significant advancements in terms of the Edit Progress metric for the CRA task. For instance, on the $T5\text{-}Review_{data}$ dataset, `CodeReviser` achieves an EP value of 31.3%, representing an increase of around 20% compared to the state-of-the-art CodeReviewer (31.3% vs. 26.3%). Also, our statistical test results show that the p-values comparing the Edit Progress values of `CodeReviser` and those achieved by the baselines are all less than 0.01 in the two datasets.

By comparing the results listed in Tables 2 and 3, we note that all the code revision approaches achieve higher performances on the CRA task compared with the CRB task. This implies that incorporating reviewer's comments as guidance in code revision tasks can result in higher quality revision outcomes. Upon manual examination, we observe that reviewer's comments often offer clear instructions on how to carry out code revisions, without which the model might perform common but incorrect code transformations. Listing 4 gives a concrete example. In this case, the reviewer's comment, "Return quickFilters", likely suggests a code revision to standardize the coding style across the entire project by unifying the way of invoking class fields. The comment provides highly specific guidance on how to perform the revision, and when incorporated into the approach input, `CodeReviser` successfully produces the desired code, indicating the effectiveness of this guidance. However, without the comment, `CodeReviser` changes the return type of this method from `List` to `Collection`. The intent behind this code change may be to utilize a more abstract interface type as the return type, thereby reducing dependencies on specific implementation classes. This modification offers the potential advantage that if there is a need to substitute the specific collection implementation in the future, one would only have to adjust the code within the method, leaving the code that calls it unaffected. As a result, altering the return type in this scenario could enhance the flexibility, extensibility, and maintainability of the code. It is worth noting that updating the return type, known as *Return Type Update* is a common practice in software maintenance [12], which could be a reason for `CodeReviser` to learn to perform this type of revision.

6.3 RQ3: Localization Accuracy

The localization accuracy of different approaches is shown in Table 4. We observe that the localization capability of `CodeReviser` exceeds those of the baselines to a large margin. Specifically, when performing the CRB task, the localization accuracy of `CodeReviser` reaches 26.1% and 12.3% on the $Trans\text{-}Review_{data}$ and $T5\text{-}Review_{data}$ datasets, respectively, outperforming the best performances

```
// Oracle Code Revision
public List<QuickFilter> getQuickFilters() {
-    return this.quickFilters;
+    return quickFilters;
}

// Code Revision by CodeReviser when the reviewer's comment is unavailable
- public List<QuickFilter> getQuickFilters() {
+ public Collection<QuickFilter> getQuickFilters() {
     return this.quickFilters;
}
```

Listing 4. The oracle code revision and the revision generated by `CodeReviser` when the reviewer's comment is unavailable for the method `getQuickFilters`.

Table 4. The localization accuracy of different approaches (in %).

Task	Dataset	CodeT5	NatGen	CoditT5	CodeReviewer	CCT5	`CodeReviser`
CRB	*Trans-Review*$_{data}$	11.9	15.8	13.5	17.3	18.6	**26.1**
	T5-Review$_{data}$	7.7	7.3	9.4	9.9	9.8	**12.3**
CRA	*Trans-Review*$_{data}$	40.3	38.7	41.8	40.5	42.5	**47.1**
	T5-Review$_{data}$	25.0	25.0	24.5	26.5	25.7	**30.5**

Table 5. The revision accuracy of different approaches (in %).

Task	Dataset	CodeT5	NatGen	CoditT5	CodeReviewer	CCT5	`CodeReviser`
CRB	*Trans-Review*$_{data}$	71.8	67.8	66.1	71.6	72.6	**74.2**
	T5-Review$_{data}$	61.4	61.2	58.9	56.6	57.8	**61.7**
CRA	*Trans-Review*$_{data}$	74.9	84.7	81.4	81.1	82.1	**86.7**
	T5-Review$_{data}$	63.6	66.3	66.9	62.6	65.3	**67.2**

from existing approaches by around 40% (26.1% vs. 18.6%) and 25%(12.3% vs. 9.9%). In terms of the CRA task, the localization accuracy of `CodeReviser` increases to 47.1% and 30.5% on the two datasets respectively, exceeding the best-performing baselines by around 11% (47.1% vs. 42.5%) and 15% (30.5% vs. 26.5%). The obtained results provide evidence that our approach excels in identifying the code segments that require revisions, which may be attributed to the utilization of a specialized localizer specifically designed for this task. By customizing the localizer to focus on the accurate localization, our approach demonstrates enhanced performances in identifying the areas of code that necessitate revisions.

6.4 RQ4: Revision Accuracy

The revision accuracy of different approaches is listed in Table 5. We note that `CodeReviser` still consistently outperforms the existing approaches in terms of the revision accuracy under different settings. For instance, when performing the CRA task, the revision accuracy of `CodeReviser` reaches 86.7% and 67.2% on the *Trans-Review*$_{data}$ and *T5-Review*$_{data}$ datasets, respectively, while the best performances from the baselines are 84.7% and 66.9%. Such results suggest that `CodeReviser` is more capable of generating correct revisions at the ground-truth locations, compared with existing approaches. We also investigate the values of CodeBLEU and Edit Progress of different approaches on the test samples whose ground truth locations are already identified. Such results are listed in Table 6 and Table 7. We observe the same phenomenon as that from Table 5, i.e., compared with existing approaches, `CodeReviser` achieves better effectiveness in terms of CodeBLEU and Edit Progress on the test samples whose ground truth locations are already identified. For instance, regarding the CRB task, the CodeBLEU value of `CodeReviser` reaches 91.6% and 89.5% on the

Table 6. The CodeBLEU of different approaches when the ground truth locations are already identified (in %).

Task	Dataset	CodeT5	NatGen	CoditT5	CodeReviewer	CCT5	`CodeReviser`
CRB	$Trans\text{-}Review_{data}$	90.3	89.7	88.1	90.9	90.5	**91.6**
	$T5\text{-}Review_{data}$	86.4	86.2	85.6	85.7	86.4	**89.5**
CRA	$Trans\text{-}Review_{data}$	91.4	94.2	93.0	93.4	93.5	**94.8**
	$T5\text{-}Review_{data}$	90.2	90.0	91.0	90.0	90.6	**91.2**

Table 7. The Edit Progress of different approaches when the ground truth locations are already identified (in %).

Task	Dataset	CodeT5	NatGen	CoditT5	CodeReviewer	CCT5	`CodeReviser`
CRB	$Trans\text{-}Review_{data}$	79.4	75.7	73.6	79.3	80.1	**80.2**
	$T5\text{-}Review_{data}$	70.3	70.3	68.7	66.0	65.9	**70.4**
CRA	$Trans\text{-}Review_{data}$	83.9	89.2	87.4	86.9	86.2	**90.7**
	$T5\text{-}Review_{data}$	74.3	72.9	75.8	72.9	74.0	**76.7**

$Trans\text{-}Review_{data}$ and $T5\text{-}Review_{data}$ datasets, respectively, while the best performances from the baselines are 90.9% and 86.4%.

By comparing the figures in Tables 4 and 5, we have made two key observations. First, the revision accuracy of `CodeReviser` significantly surpasses its localization accuracy. This may indicate that the localizer of `CodeReviser` may be the main performance bottleneck for the overall approach. To gain a quantitative understanding of this observation, we conduct an experiment on the large-scale $T5\text{-}Review_{data}$ dataset in which we provide the reviser with the oracle localization results and evaluate its performance. The results reveal that under these ideal conditions, the reviser achieves an Exact Match value of 31.0%, surpassing the current value of 20.5% by over 50%. This finding indicates that improving the effectiveness of the localizer is likely to have a significant impact on the overall performance of the approach, and therefore, future efforts could be devoted into this direction. Second, the improvement of `CodeReviser` over the baselines in terms of revision accuracy is not as significant as its improvement in terms of localization accuracy. One possible explanation is that existing approaches have already acquired substantial knowledge about code revisions during their pre-training processes, which may narrow the performance gap between them and `CodeReviser`. For instance, NatGen is specifically trained to transform unnatural code into its formal version, which likely enhances its capability for code revision. Consequently, when applied to the CRB task, NatGen achieves a revision accuracy of 61.6% on the $T5\text{-}Review_{data}$ dataset, which is only slightly lower than that of `CodeReviser`.

7 DISCUSSION

7.1 Is Precise Localization a Prerequisite for Correct Revision?

Since `CodeReviser` comprises a localizer and a reviser, a relevant question arises: *is precise localization a prerequisite for correct revision*? To address this, we explore whether the reviser can produce the correct code with inaccurate localization information. The answer, it turns out, is affirmative. For instance, in the $T5\text{-}Review_{data}$ dataset, there are 295 cases where the localizer initially fails to precisely identify the parts requiring revisions but the reviser ultimately produces an exact match (w.r.t the CRA task). Such results reveal the actual localization capability of the localizer might be slightly lower than the figures shown in Table 4.

A concrete example is shown in Listing 5. In this case, the reviewer asks for a code change by commenting "Maybe call expectThrowable for consistency?", indicating that within the body of the revised method, it should invoke an overloaded method with the same name. To make this revision, the developer simply needs to modify the name of the invoked method. As demonstrated

```
// Oracle Code Revision
public static ExceptionThrowingSubTest expectThrowable(Runnable runnable) {
-    return expectException(runnable.toString(), runnable);
+    return expectThrowable(runnable.toString(), runnable);
}
```

```
// Input code token sequence
... return expectException ( runnable . toString ( ) , runnable ) ; }
// Oracle localization result
... return [START] expectException [END] ( runnable . toString ( ) , runnable ) ; }
// Localization result from the localizer
... return [START] expectException ( runnable . toString ( ) , runnable ) ; [END] }
```

Listing 5. An example where the localizer of CodeReviser does not generate an accurate localization result but the reviser finally produces the oracle revision.

in the second part of the listing, the oracle localization result is only associated with such a single token. On the contrary, CodeReviser's localizer identifies the entire expression (highlighted in grey) as contents that require revisions, resulting in an inaccurate localization outcome for the reviser. Nevertheless, CodeReviser's reviser still produces the correct ground-truth code. This example highlights the reviser's ability to overcome inaccurate localization results, demonstrating that it can generate accurate revisions even when the localization is not precise.

The example presented in this section aims to showcase that CodeReviser can generate accurate revisions even with imprecise localization. In other words, the localisation information given by the localiser does not limit the location of code revision, and the reviser has a recovery capacity if wrong localisation information is given.

7.2 Comparison with Large Language Models

Recently, various large language models (LLMs) have been proposed to facilitate developers' development activities, such as ChatGPT[9] and GPT-4[10]. A number of studies have showcased that LLMs can achieve competitive results through prompting when compared with the state-of-the-art approaches in tasks like code summarization [15], code generation [9], and test generation [8]. Therefore, it is crucial to evaluate the performance of LLMs in the context of code revision tasks. To that end, we perform experiments from two perspectives to understand how well the LLMs can achieve on the code revision task: we first directly prompt a commercial LLM, after which we fine-tune an open source LLM to make it be aware of the domain knowledge of code revision.

7.2.1 *Prompting of the Commercial LLM.*

Considering the costs of prompting a commercial LLM, we perform an experiment on the test set of the large-scale $T5\text{-}Review_{data}$ dataset to quantitatively understand the code revision effectiveness of ChatGPT. Specifically, we use the ChatGPT API (accessed on November 5, 2023, and the temperature is set to 0) with the prompts to the model in the form of:

[9] https://chat.openai.com/
[10] https://openai.com/product/gpt-4

```
Assume that you are a Java programmer.
Below is a code snippet that may contain some syntax or semantic errors:
# <CODE>
The comment from the code reviewer is:
# <COMMENT>
Please help me revise the code and output a new version.
You can only show me the revised code.
```

The first sentence is to prepare ChatGPT for the task related to Java language, followed by the detailed task. The last sentence is to restrict ChatpGPT to only output the code. Note that for the code revision before review task, the review comment information is removed from the prompt. We calculate the metrics for the revised code generated by ChatGPT. Results show that for the code revision before review task, the performances of ChatGPT are: 0.8% of Exact Match, 59.6% of CodeBLEU, and 3.2% of Edit Progress; for the code revision after review task, the performances of ChatGPT are: 9.3% of Exact Match, 70.2% of CodeBLEU, and 17.2% of Edit Progress. Compared with the results listed in Tables 2 and 3, it is worth noting that ChatGPT, when utilized in the zero-shot manner, does not outperform the state-of-the-art code revision approaches. For instance, on the code revision after review task, `CodeReviser` can generate oracle code for more than 20% cases, while the percentage of ChatGPT is less than 10%.

7.2.2 Fine-tuning of the Open Source LLM. A recent study has proposed an approach, LLaMA-Reviewer [33], to fine-tune the LLMs on the code revision task, and in our study, we decide to reproduce their approach on our test datasets. Specifically, the utilized LLM is LLaMA [11] and the fine-tuning process adopts a parameter-efficient way, i.e., the LoRA approach that contains less than 1% of trainable parameters compared with the original model. We fine-tune LLaMA under the four settings (two datasets × two tasks) and the results are shown in Table 8. Results show that our `CodeReviser` systematically outperforms LLaMA-Reviewer with respect to all the metrics. Specifically, LLaMA-Reviewer achieves similar CodeBLEU values to `CodeReviser`, but it is significantly outperformed by `CodeReviser` on the other two metrics, particularly on the $T5\text{-}Review_{data}$ dataset (e.g., 31.3% v.s. 9.8% in terms of the Edit Progress achieved on the CRA task).

Table 8. The Comparison Between `CodeReviser` and LLaMA-Reviewer (in %).

Task	Dataset	LLaMA-Reviewer			`CodeReviser`		
		Exact Match	CodeBLEU	Edit Progress	Exact Match	CodeBLEU	Edit Progress
CRB	$Trans\text{-}Review_{data}$	6.2	72.3	42.2	19.3	73.4	48.4
	$T5\text{-}Review_{data}$	1.7	70.7	3.9	7.6	74.1	12.8
CRA	$Trans\text{-}Review_{data}$	22.0	78.6	54.1	40.8	80.0	64.4
	$T5\text{-}Review_{data}$	6.0	75.6	9.8	20.5	76.4	31.3

Our investigations in this section reveals that both directly prompting or fine-tuning the LLMs may not achieve the optimal results compared with the state-of-the-art approaches, underscoring the need for further explorations into the utilization of LLMs for code revision tasks, which we plan to pay attention to in our future work.

7.3 Automated Code Revision v.s. Automated Program Repair

Another long-studied task in the software engineering domain, automated program repair [4, 10, 19, 30, 31, 64], also involves automatically changing the source code of a program (i.e., modifying the buggy code into a correct version). This task, however, is mainly different from our study subject, automated code revision, in the following two aspects. From the code specification perspective,

[11] https://llama.meta.com/

automated code revision approaches modify the code to implement the requirements described by the review comments, which are in the natural language form; while automated program repair approaches try to generate patches that can pass all the test cases, which are in the programming language form. This means that automated program repair approaches can adopt a try-and-error strategy to generate patches and finally output a test-adequate patch, while the patch generation process of automated code revision approaches is one-shot since there is no clear indicator for the correctness of the patches. From the approach input perspective, automated code revision approaches usually take a function plus with review comments as the inputs, since they are unaware of the specific buggy locations; while thanks to the test coverage analysis, automated program repair approaches often take a buggy statement as the input, which is a finer-grained granularity. Despite the differences, our "Divide-and-Conquer" approach design is inspired by the existing automated program repair techniques, which usually have a fault localization step that identifies the buggy statement from the entire code project and then modify the buggy statement to generate patches.

7.4 Threats to Validity

External threats. Recent years have witnessed a large number of pre-trained models being proposed [39]. Our study may have a selection bias by considering several of them as baselines. However, the involved pre-trained models are the most popular ones and have been shown to be the state-of-the-art approaches on code revision tasks [68].

Another threat is from the dataset perspective. In this study, we choose two widely-used datasets in the code review domain to evaluate the effectiveness of `CodeReviser`. These two datasets only contain code review data from Java programming language. In our future work, we plan to extent the evaluation to other languages, probably by reusing another dataset released by the previous study [27].

Internal threats. The evaluation metrics may influence the experiment results as well as the reached conclusions of the study. To mitigate the bias incurred by the metric selection, we use three metrics in our evaluation that assess the quality of the generated code from different perspectives.

`CodeReviser` builds on a pre-trained model (i.e., CodeT5) which is pre-trained on extensive code-related data. There is thus a risk of data leakage. To investigate such a situation, we have checked the evaluation datasets in our study and the pre-training dataset of CodeT5, and confirmed that there is no overlapped data, meaning that our study does not face the threat from data leakage.

Another threat comes from the dataset splitting. The two evaluation datasets (i.e., $\textit{Trans-Review}_{data}$ and $\textit{T5-Review}_{data}$) are split into training, validation, test sets in an 8:1:1 ratio by the original studies [53, 54]. Although such splittings are widely adopted by recent studies, the effect of such a splitting to the overall performance is unknown. To better investigate this question, we perform 10-fold cross validation [22, 61] to evaluate the effectiveness of `CodeReviser` on the CRB task. Results show that `CodeReviser` achieves rather similar performances in each round. Specifically, the Exact Match of `CodeReviser` ranges from 19.1% to 19.6% on the $\textit{Trans-Review}_{data}$ dataset, while such a value ranges from 7.3% to 7.7% on the $\textit{T5-Review}_{data}$ dataset. Such results indicate that the dataset splitting has a negligible impact on the overall performance. Therefore, in this paper, we report the performances of different approaches achieved on the common dataset splitting.

8 RELATED WORK

8.1 Automating Other Code Review Activities

In this study, we focus on automating code revisions during the code review process. In the literature, there are also a number of studies automating other code review activities. For instance, studies have proposed to reduce the workload of developers on writing reviews [17, 47]. Specifically, CORE [47] uses a multi-level embedding approach, which focuses on both word-level and character-level information of code tokens, to embed code changes and reviews. After that, an attention neural network is used to predict the relevance score between a code change and a review, and the review with the highest score will be recommended. CommentFinder [17] is also a retrieval-based approach which does not require any deep learning model. Given a changed method, it searches for the most similar changed methods in a large-scale corpus through a two-step similarity measurement and reuses the associated reviews. Recent studies have also proposed to utilize pre-trained models to generate reviews directly based on the submitted code [27, 53].

Reviewer recommendation is also a hot topic in this domain since the previous study finds that a non-negligible proportion of pull requests cannot be assigned to appropriate reviewers, which negatively results in longer reviewing time [50]. CORRECT considers not only the relevant cross-project work experience of a developer but also his/her experience in certain specialized technologies when determining the potential reviewer [41]. The previous study [65] proposes to construct a novel *Comment Network* by mining historical commenting interactions between code authors and code reviewers, and predict highly relevant reviewers based on this social network. The behind intuition is that developers who share common interests with the authors could be appropriate reviewers.

8.2 Divide-and-Conquer in Software Engineering

As a foundational principle in software engineering, "Divide-and-Conquer" has been widely applied in researches from this domain. According to whether the updated contents can be found from the code change, Toper splits the just-in-time comment updating task into two sub-tasks (i.e., code-indicative updates and non-code-indicative updates) and respectively utilizes a heuristic-based approach and a deep-learning-based model to effectively address the sub-tasks [29]. Traditional translation-based code migration approaches often produce syntactically incorrect code [37]. To overcome this limitation, mppSMT splits the task into multiple phases where the syntactic structure of the target code is generated first and the lexical tokens (e.g., APIs) are filled later [38]. Similarly, SkCoder employs a retrieval-and-edit paradigm to accomplish the task of automatic code generation (i.e., generating code snippets that fulfill the given natural language descriptions), instead of generating the code from scratch [26]. Specifically, the process involves utilizing a *retriever* to select a code snippet from a corpus as a preliminary sketch, followed by employing an *editor* to modify the sketch and obtain the final target code. These studies motivate us to explore a code revision approach that follows the "Divide-and-Conquer" principle.

9 CONCLUSION

This study introduces a two-phase code revision approach, designed to address the labor-intensive nature of code review. By adhering to the "Divide-and-Conquer" principle, we separate the tasks of localization and revision, which allows us to focus on each task individually and develop targeted solutions for more effective code revision. Our approach, consisting of a localizer and a reviser, outperforms existing approaches in extensive experiments. Specifically, when revising code based on reviewer's comments, our approach demonstrates a remarkable success rate of over 20% to generate the oracle revision, surpassing the state-of-the-art approaches by a significant margin. These results underscore the effectiveness of our approach in automating the code revision activities during code review processes. Our analysis reveals that boosting the localization ability is a promising direction for improving the effectiveness of code revision approaches. Technical strategies that can be foreseen include utilizing more advanced code models and exploring the semantic relationships between the review comments and the code.

10 DATA AVAILABILITY

All code and data in this study are publicly available at:

https://zenodo.org/record/8373320

ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China (No. 2022YFA1005101), the National Natural Science Foundation of China No.62402506, as well as the Research Foundation from NUDT (Grant No. ZK24-05).

REFERENCES

[1] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 712–721. https://doi.org/10.1109/ICSE.2013.6606617

[2] Gabriele Bavota and Barbara Russo. 2015. Four eyes are better than two: On the impact of code reviews on software quality. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 81–90. https://doi.org/10.1109/ICSM.2015.7332454

[3] Amiangshu Bosu and Jeffrey C Carver. 2013. Impact of peer code review on peer impression formation: A survey. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 133–142. https://doi.org/10.1109/ESEM.2013.23

[4] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. *arXiv preprint arXiv:2403.17134* (2024).

[5] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar Devanbu, and Baishakhi Ray. 2022. NatGen: Generative pre-training by "Naturalizing" source code. In *Proceedings of the 30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM. https://doi.org/10.1145/3540250.3549162

[6] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. 2015. Code reviews do not find bugs. how the current code review best practice slows us down. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 27–28. https://doi.org/10.1109/ICSE.2015.131

[7] Florian Deissenboeck and Markus Pizka. 2006. Concise and consistent naming. *Software Quality Journal* 14 (2006), 261–282. https://doi.org/10.1007/s11219-006-9219-1

[8] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 423–435. https://doi.org/10.1145/3597926.3598067

[9] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration Code Generation via ChatGPT. *arXiv preprint arXiv:2304.07590* (2023).

[10] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481. https://doi.org/10.1109/ICSE48619.2023.00128

[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. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[12] Beat Fluri and Harald C Gall. 2006. Classifying change types for qualifying change couplings. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, 35–45. https://doi.org/10.1109/ICPC.2006.16

[13] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[14] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Shaomeng Cao, Kechi Zhang, and Zhi Jin. 2023. Interpretation-based code summarization. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE, 113–124. https://doi.org/10.1109/ICPC58990.2023.00026

[15] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3597503.3608134

[16] 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. GraphCode{BERT}: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*. https://openreview.net/forum?id=jLoC4ez43PZ

[17] Yang Hong, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, and Aldeida Aleti. 2022. Commentfinder: a simpler, faster, more accurate code review comments recommendation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 507–519. https://doi.org/10.1145/3540250.3549119

[18] 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).

[19] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 1646–1656. https://doi.org/10.1145/3611643.3613892

[20] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 23rd International Symposium on Software Testing and Analysis*. ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[21] Donald Ervin Knuth. 1997. *The art of computer programming*. Vol. 3. Pearson Education.

[22] Ron Kohavi. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Fourteenth International Joint Conference on Artificial Intelligence*. Montreal, Canada, 1137–1145.

[23] Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Kui Liu, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2019. D&c: A divide-and-conquer approach to ir-based bug localization. *arXiv preprint arXiv:1902.02703* (2019).

[24] Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. Soviet Union, 707–710.

[25] Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. 2021. Editsum: A retrieve-and-edit framework for source code summarization. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 155–166. https://doi.org/10.1109/ASE51524.2021.9678724

[26] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. SkCoder: A Sketch-based Approach for Automatic Code Generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2124–2135. https://doi.org/10.1109/ICSE48619.2023.00179

[27] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating Code Review Activities by Large-Scale Pre-Training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. 1035–1047. https://doi.org/10.1145/3540250.3549081

[28] Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. 2023. CCT5: A Code-Change-Oriented Pre-Trained Model. In *Proceedings of the 31st ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM. https://doi.org/10.1145/3611643.3616339

[29] Bo Lin, Shangwen Wang, Zhongxin Liu, Xin Xia, and Xiaoguang Mao. 2022. Predictive comment updating with heuristics and ast-path-based neural learning: A two-phase approach. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1640–1660. https://doi.org/10.1109/TSE.2022.3185458

[30] Bo Lin, Shangwen Wang, Ming Wen, Liqian Chen, and Xiaoguang Mao. 2024. One Size Does Not Fit All: Multi-granularity Patch Generation for Better Automated Program Repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/3650212.3680381

[31] Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. 2022. Context-aware code change embedding for better patch correctness assessment. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–29. https://doi.org/10.1145/3505247

[32] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the Efficiency of Test Suite based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *Proceedings of the 42nd International Conference on Software Engineering*. ACM, 615–627. https://doi.org/10.1145/3377811.3380338

[33] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 647–658. https://doi.org/10.1109/ISSRE59848.2023.00026

[34] Laura MacLeod, Michaela Greiler, Margaret-Anne Storey, Christian Bird, and Jacek Czerwonka. 2017. Code reviewing in the trenches: Challenges and best practices. *IEEE Software* 35, 4 (2017), 34–42. https://doi.org/10.1109/MS.2017.265100500
[35] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2014. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th working conference on mining software repositories*. 192–201. https://doi.org/10.1145/2597073.2597076
[36] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. 2015. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE, 171–180. https://doi.org/10.1109/SANER.2015.7081827
[37] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 651–654. https://doi.org/10.1145/2491411.2494584
[38] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2015. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 585–596. https://doi.org/10.1109/ASE.2015.74
[39] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An Empirical Comparison of Pre-Trained Models of Source Code. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2136–2148. https://doi.org/10.1109/ICSE48619.2023.00180
[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. *Journal of Machine Learning Research* 21 (2020), 1–67. https://doi.org/10.5555/3455716.3455856
[41] Mohammad Masudur Rahman, Chanchal K Roy, Jesse Redl, and Jason A Collins. 2016. Correct: Code reviewer recommendation at github for vendasta technologies. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 792–797. https://doi.org/10.1145/2970276.2970283
[42] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).
[43] Peter C Rigby and Christian Bird. 2013. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. 202–212. https://doi.org/10.1145/2491411.2491444
[44] Peter C Rigby, Daniel M German, Laura Cowen, and Margaret-Anne Storey. 2014. Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4 (2014), 1–33. https://doi.org/10.1145/2594458
[45] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*. 181–190. https://doi.org/10.1145/3183519.3183525
[46] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *54th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics (ACL), 1715–1725.
[47] Jing Kai Siow, Cuiyun Gao, Lingling Fan, Sen Chen, and Yang Liu. 2020. Core: Automating review recommendation for code changes. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 284–295. https://doi.org/10.1109/SANER48275.2020.9054794
[48] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 130–140. https://doi.org/10.1109/SANER.2018.8330203
[49] Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. 2022. Autotransform: Automated code transformation to support modern code review process. In *Proceedings of the 44th International Conference on Software Engineering*. 237–248. https://doi.org/10.1145/3510003.3510067
[50] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 141–150. https://doi.org/10.1109/SANER.2015.7081824
[51] 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. https://doi.org/10.1109/ICSE.2019.00021
[52] 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* 28, 4 (2019), 19:1–19:29. https://doi.org/10.1145/3340544

[53] 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. https://doi.org/10.1145/3510003.3510621

[54] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. 2021. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 163–174. https://doi.org/10.1109/ICSE43902.2021.00027

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. 30 (2017). https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[56] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer Networks. In *Advances in Neural Information Processing Systems*, Vol. 28. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2015/file/29921001f2f04bd3baee84a12e98098f-Paper.pdf

[57] Shangwen Wang, Mingyang Geng, Bo Lin, Zhensu Sun, Ming Wen, Yepang Liu, Li Li, Tegawendé F. Bissyandé, and Xiaoguang Mao. 2023. Natural Language to Code: How Far are We?. In *Proceedings of the 31st ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM. https://doi.org/10.1145/3611643.3616323

[58] Shangwen Wang, Bo Lin, Zhensu Sun, Ming Wen, Yepang Liu, Yan Lei, and Xiaoguang Mao. 2023. Two birds with one stone: Boosting code generation and code search via a generative adversarial network. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 486–515. https://doi.org/10.1145/3622815

[59] Shangwen Wang, Kui Liu, Bo Lin, Li Li, Jacques Klein, Xiaoguang Mao, and Tegawendé F Bissyandé. 2021. Beep: Fine-grained fix localization by learning to predict buggy code elements. *arXiv preprint arXiv:2111.07739* (2021).

[60] Shangwen Wang, Ming Wen, Bo Lin, Yepang Liu, Tegawendé F. Bissyandé, and Xiaoguang Mao. 2023. Pre-implementation Method Name Prediction for Object-oriented Programming. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 157 (sep 2023), 35 pages. https://doi.org/10.1145/3597203

[61] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated Patch Correctness Assessment: How Far are We?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia). ACM, 968–980. https://doi.org/10.1145/3324884.3416590

[62] 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. https://doi.org/10.18653/v1/2021.emnlp-main.685

[63] F. Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.

[64] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[65] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. 2016. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? *Information and Software Technology* 74 (2016), 204–218. https://doi.org/10.1016/j.infsof.2016.01.004

[66] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An Extensive Study on Pre-trained Models for Program Understanding and Generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM. https://doi.org/10.1145/3533767.3534390

[67] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2022. CoditT5: Pretraining for Source Code and Natural Language Editing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ACM. https://doi.org/10.1145/3551349.3556955

[68] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, Junda He, and David Lo. 2023. Generation-based Code Review Automation: How Far Are We?. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. 215–226. https://doi.org/10.1109/ICPC58990.2023.00036

[69] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2892102