# Automated Code Editing with Search-Generate-Modify

Changshu Liu, Pelin Cetin*, Yogesh Patodia*, Baishakhi Ray, Saikat Chakraborty, Yangruibo Ding

***Abstract*—Code editing is essential in evolving software development. In literature, several automated code editing tools are proposed, which leverage Information Retrieval-based techniques and Machine Learning-based code generation and code editing models. Each technique comes with its own promises and perils, and for this reason, they are often used together to complement their strengths and compensate for their weaknesses. This paper proposes a hybrid approach to better synthesize code edits by leveraging the power of code search, generation, and modification.**

**Our key observation is that a patch that is obtained by search & retrieval, even if incorrect, can provide helpful guidance to a code generation model. However, a retrieval-guided patch produced by a code generation model can still be a few tokens off from the intended patch. Such generated patches can be slightly modified to create the intended patches. We developed a novel tool to solve this challenge: SARGAM, which is designed to follow a real developer's code editing behavior. Given an original code version, the developer may *search* for the related patches, *generate* or write the code, and then *modify* the generated code to adapt it to the right context. Our evaluation of SARGAM on edit generation shows superior performance w.r.t. the current state-of-the-art techniques. SARGAM also shows its effectiveness on automated program repair tasks.**

***Index Terms*—Bug fixing, Automated Program Repair, Edit-based Neural Network**

## I. INTRODUCTION

In a rapidly-evolving software development environment, developers often edit code to fix bugs, add new features, or optimize performance. This process can be complex and requires a deep understanding of the underlying programming language, as well as an expertise in the relevant domain. To facilitate code editing, developers often search existing codebases [1–3] or online resources [4] for relevant code, and may also leverage automated code generation tools such as GitHub Copilot[1]. However, the search results [5, 6] or generated code may not always be ideal, necessitating developers to customize them for the given situation [7]. Therefore, developers may have to further modify the generated code to achieve the desired outcome.

In the past, various tools and techniques have been proposed to reduce the manual effort required for code editing [8–11]. They can be broadly classified into three different categories: Search & Retrieve, Generate, and Modify.

*Search & Retrieve.* This is a popular approach to suggest edits that were previously applied to similar code contexts [1, 2, 12, 13]. However, each retrieval-based technique relies on its perceived definition of code similarity (*e.g.,* token, tree, or graph-based similarity) and fails to generate edits with a slight variation of that definition. As a result, these methods tend to have limited applicability to diverse code editing contexts.

*Generate.* In recent years, the most promising approach is perhaps the Large Language Model (LLM)-based code generation models where code is generated based on developers' intent and surrounding code context. For instance, open-source code-specific LLMs such as PLBART [14], CodeGPT-2 [15], CodeT5 [16], and NatGen [17] have shown significant potential in code generation. Additionally, industry-scale LLMs like GPT-3 [18] and Codex [19] have gained widespread popularity for generating source code and are used as the backbone of commercial code generation software such as GitHub Copilot[1].

There is a subtle difference between edit generation and code generation. Developers generate edits by transforming the original code version into a new version, often by deleting and adding lines. Edit generation can thus be thought of as a conditional probability distribution, generating new lines of code by conditioning on the old lines. Existing LLM-based code generation approaches do not capture granular edit operations: which tokens will be deleted, which tokens will be inserted, and in particular, where the new tokens will be inserted.

*Modify.* Many previous works [20–23] designed special outputs to represent edit operations. Recently CoditT5 [23] proposes an edit-specific LLM where given an original code version, CoditT5 [23] first comes up with an edit plan (in terms of deletion, insertion, substitution) and then conditioned on the edit plan, it generates the edits in an auto-regressive manner. CoditT5 [23] shows promise in generating edits over vanilla code generation.

The goal of this work is to produce higher-quality code edits by harnessing the power of all three techniques. Each approach offers unique ingredients that can contribute to better edit generation.

**Our Insight.** Code search can retrieve relevant patches that can provide more guidance to a code generation model, leading to better patch generation. However, most of the time the patches generated this way are off by a few tokens from the intended patch—even random permutations and combinations of the generated tokens could lead to the intended patch [24]. A more systematic approach would involve using an edit-

Changshu Liu, Pelin Cetin, Yogesh Patodia, Baishakhi Ray and Yangruibo Ding are affiliated with Department of Computer Science, Columbia University, New York, NY USA.
Email: {cl4062@, pc2807@, yp2607@, rayb@cs., yrbding@cs.}columbia.edu
Saikat Chakraborty is affiliated with Microsoft Research Redmond, WA, USA.
Email: saikatc@microsoft.com
Manuscript received May, 2023; revised Feb, 2024.
[1]https://github.com/features/copilot/

generation model that specifically targets the generated tokens that require further modifications such as deletion or insertion. This allows more focused and precise modifications of the code generated in the previous step and finally outputs the intended patch.

**Proposed Approach.** We propose a framework, SARGAM, that leverages code-search-augmented code generation and modification to generate code edits. SARGAM emulates the typical code editing practice of a developer where given an edit location and context, she might search for related code, write the retrieved code (*i.e.,* generation) to the target location, and modify it to contextualize. SARGAM contains three steps: (i) Search: An information retrieval-based technique to retrieve candidate patches from a database of previous edits that may fit the edit context, (ii) Generation: An off-the-shelf code generation model that takes input from the edit location, edit context, and the retrieved patches, and outputs a token sequence corresponding to the edited code, and (iii) Modification: A novel code editing model that slightly modifies the token sequence generated in the previous step and outputs granular edit operations in terms of deleted and inserted tokens.

As opposed to the existing edit-generation models [23] that aim to generate the edit operations directly from the original version, we allow a generation model to initially generate the token sequence and then refine it to produce the final patch. We observe that a granular edit model generally performs better for generating smaller edits. If a generation model already generates a sufficiently accurate patch, enhancing it with further edits can improve the overall effectiveness of the edit-generation model.

**Results.** We evaluate our approach on two tasks: code editing and program repair. For code editing, we examine SARGAM on two different datasets. SARGAM improves $top\ 1$ patch generation accuracy over state-of-the-art patch generation models (PLBART [14], NatGen [17] and CoditT5 [23]) from 19.76% to 2.77% in different settings. For program repair, we compare SARGAM with recent Deep Learning-based techniques on Defects4J$_{1.2}$, Defects4J$_{2.0}$, and QuixBugs datasets and report state-of-the-art performance. Additionally, we conduct extensive ablation studies to justify our different design choices. In particular, we investigate three components (search, generate, and modify) individually and prove that SARGAM can benefit from each one of them.

In summary, our key contributions are:

- We prototype a code editing model, SARGAM, built on top of off-the-shelf pre-trained code generation models and augmented the generation model with code search and code modification.
- We propose a new code modification model, which involves generating granular edit operations (*i.e.,* deletion and insertion operations at token granularity as opposed to generating token sequences).
- We demonstrate SARGAM's ability to generate patches for general-purpose code edits and bug fixes. Across most of the settings, SARGAM achieves state-of-the-art performances. We present a detailed ablation study to justify our different design choices.
- We release our prototype tool at https://github.com/SarGAMTEAM/SarGAM.git.

## II. BACKGROUND: CODE GENERATION MODELS

Machine Learning-based Code Generation has gained significant attention in recent years, where code is generated from a Natural Language (NL) description or code context. Different types of Sequence-to-Sequence ($seq2seq$) models play a significant role in achieving this success [25, 26]. The input to a $seq2seq$ model is a sequence of tokens ($X = x_1, x_2, ..., x_n$), and the output is a token sequence ($Y = y_1, y_2, ..., y_m$), where the model learns conditional probability distribution $P(Y|X)$.

Recurrent Neural Networks (RNN) and Long Short Term Memory (LSTM)-based models [27] once held a dominant role in code generation [11, 28–30]. RNNs and LSTMs take a piece of code token-by-token in a sequential manner and try to predict the next token conditioned on the immediately preceding tokens in the sequence. The two types of models largely depend on the tokens in close vicinity and tend to suffer from not capturing the long-range dependencies [31].

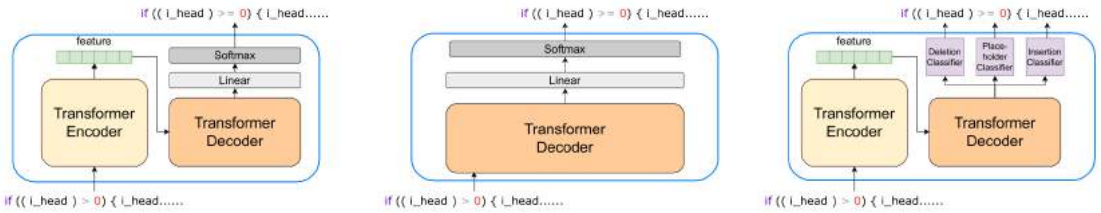
### A. Transformer for Code Generation

Transformer-based models [32] have recently outperformed alternative architectures for code generation due to the introduction of the self-attention mechanism. Transformers process the entire input token sequence as a complete graph [2]. Each token is a vertex in the graph, and an edge connecting two vertices is the "attention" between the corresponding tokens. The attention is the relative influence of a token to represent other tokens in the sequence. The attention weights signify the importance of a token to make the final prediction for a particular task [33, 34]. The model learns the attention weights depending on the task during the training process. The Transformer also encodes the relative position of each token in the input sequence (positional encoding).

The attention mechanism and positional encoding allow Transformers to catch more long-range dependencies. The self-attention mechanism allows parallel processing of input sequences that leads to significant speedup during training [32]. Many previous works use Transformers for code generation problems (*e.g.,* patching, code editing, and program repair) due to their success [17, 35–37]. Transformer-based models roughly fall into two categories: encoder-decoder and decoder-only.

**Encoder-decoder**. As shown in Figure 1a, an encoder-decoder model has a Transformer encoder and an autoregressive Transformer decoder. The encoder is trained to extract features from the input sequence. The decoder generates the next token by reasoning about the feature extracted by the Transformer encoder and previously generated tokens. PLBART [14], CodeT5 [16], and NatGen [17] are examples of encoder-decoder models trained on code corpora with denoising pre-training. CoditT5 [23] further presents a custom pre-trained model for code editing tasks using the same architecture as CodeT5 [16]. MODIT [37], on the other hand, fine-tunes PLBART [14] for code editing tasks.

[2]https://en.wikipedia.org/wiki/Complete_graph




(a) Encoder-Decoder (b) Decoder (Only) (c) Levenshtein Transformer

Fig. 1: Different Types of Transformer-based Generative Models

**Decoder-only**. Decoder-only models only have an autoregressive Transformer decoder (shown in Figure 1b). Since there is no encoder, decoder-only transformer is a "generate only" architecture. Such models are pre-trained in an unsupervised way from large corpora to build Generative Pre-trained Models (GPT). Jiang *et al.* [38] shows the effectiveness of GPT for the task of source code patching. Other representative decoder-only code generation models include Ploycoder [39], OpenAI's Codex[19], GPT-3 [18], etc. Decoder-only models are suitable for open-ended code generation, where a prompt describing the functionality is passed to the model.

### B. Levenshtein Transformer

The Transformers usually generate outputs from scratch. When there is much overlap between input and output token sequences (*e.g.,* automatic text editing where only a few tokens are changed, keeping most of the text as it is), Transformers tend to suffer [40] to preserve the unchanged tokens. Levenshtein Transformers (LevTs) [40] show promises in such cases, as they use basic edit operations such as *insertion* and *deletion* to implement granular sequence transformations. Levenshtein Distance [41] between the ground truth and the output token sequence is measured during training after each deletion or insertion. The predicted operation is chosen for the next interaction if the distance reduces.

Figure 1a and Figure 1c show architectural differences between a Transformer and a LevT. Although both share the same encoder and decoder blocks, the vanilla Transformer uses a linear layer and softmax upon stacks of decoder layers to predict the next token, while LevT uses three additional classifiers to apply edit operations. In LevT, the output of the last Transformer decoder block (*e.g.,* $h = \{h_0, h_1, \cdots, h_n\}$) is passed to following classifiers:

1) Deletion Classifier: for each token in the sequence, this binary classifier predicts whether it should be deleted(=1) or kept(=0). $\pi_\theta^{\text{del}}(h_i) = softmax(W_{del}h_i)$, where $W_{del}$ is the weight matrix of the deletion classifier.
2) Placeholder Classifier: predicts how many place holders should be inserted between any consecutive token pairs in the whole sequence. $\pi_\theta^{\text{plh}}(< h_i, h_{i+1} >) = softmax(W_{plh} \cdot concat(h_i, h_{i+1}))$ ,where $W_plh$ is the weight matrix of the placeholder classifier.
3) Insertion Classifier: for each placeholder we inserted in the p revious step, the insertion classifier predicts which token should be inserted in this position: $\pi_\theta^{\text{ins}}(h_i) = softmax(W_{ins}h_i)$ ,where $W_{ins}$ is the weight matrix of the insertion classifier.

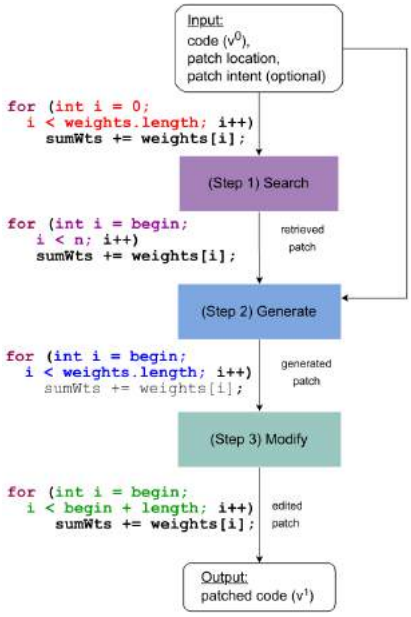



Fig. 2: Overview of the SARGAM Pipeline and a Motivating Example of a bug fixing patch taken from Defects4J$_{1.2}$ dataset. Here, inside a for loop, the loop counter initialization and loop condition ( `int i=0; i<weights.length` ) are buggy and (`int i=begin; i<begin+length`) is the expected fix. After the *Search* (Step 1), SARGAM retrieves a similar patch (`int i=begin; i<n`), the retrieval of `begin` token benefits *Generation* (Step 2). The generated patch is close to the ground truth: (`int i=begin; i<weights.length`), yet not correct. Finally, the *Modification* model (Step 3) further modifies the generated patch by deleting `weights.` and inserting `begin+`.

We choose various Transformer-based code generation models to generate patches and use a novel LevT-based edit generation model to further edit the generated patches.

## III. SARGAM APPROACH

We introduce SARGAM, a tool to synthesize source code edits (*i.e.,* patches). A patch is a set of edits to the source code used to update, fix, or improve the code. Throughout the paper, we use code edits and patches interchangeably. More formally,

*Definition 3.1:* A program patch, $p := \Delta(v^0, v^1)$, is the set of syntactic differences that update a program version $v^0$ to $v^1$. Each change is a token that is either deleted from $v^0$ or inserted to $v^1$.

We have designed SARGAM to mimic a real developer's behavior while patching source code. Given $v^0$, the developer may (i) *search* for the related patches, (ii) *generate* or write the code, and (iii) further *modify* the generated code to adapt it to the right context. To this end, we design SARGAM to have these three steps: Search, Generate, and Modify. An overview of SARGAM 's workflow is shown in Figure 2.

### A. Overview

SARGAM takes the following as input: $v^0$, the exact edit location (can be captured using the current cursor location), and optional edit intent in the form of NL. SARGAM then proceeds through the Search, Generate, and Modify steps, ultimately producing the final code version, $v^1$.

- *Step 1. Search*: Given the input as a query, SARGAM searches a database of patches to find similar patches applied previously in a similar context. This step is similar to a search-based patch recommendation engine [1]. Each retrieved patch is concatenated with the original input and passed to the next step (see Figure 3). In the motivating example in Figure 2, given the buggy code as query, the search retrieves a similar patch from the code base: for (int i=begin; i<n; i++). Although the retrieved patch is not perfect, the introduction of the <begin> token facilitates the final result.
- *Step 2. Generate.* This step takes the search augmented input and outputs a token sequence to generate the patched code. We use off-the-shelf $seq2seq$ models [14, 17, 37], as discussed in Section II-A, to generate code. Figure 2 shows the generation step produces a token sequence `for (int i=begin; i<weights.length; i++)`, which is close to the intended patch.
- *Step 3. Modify.* However, the generated patch can still be incorrect, as shown in our running example — often, they are quite close to the intended edit (*i.e.,* low edit distance), nevertheless, incorrect [42]. Developers still need to modify the generated patch here and there to get the intended output. In this step, we aim to capture such small modifications by explicitly modeling them as deleted and added operations. Our key insight is, as there is a significant overlap between the source and target token sequences, learning granular edit operations can be beneficial. In particular, we use LevT, as described in Section II-B, to explicitly model the edits as a sequence of deleted and added tokens. In the case of Figure 2, this step explicitly deletes weights. and adds token sequence begin+, resulting in the correct patch.

In this work, we implemented our own Search and Edit Models on top of existing generation models [14, 17, 23, 37]. The rest of this section elaborates each part in detail.

### B. Input Processing

While pre-processing the inputs, following MODIT [37], we create a multi-modal input capturing (i) exact patch location, (ii) patch context, (iii) developers' intent or guidance in the form of natural text. Figure 3 provides an example input. Following some recent code editing and program repair

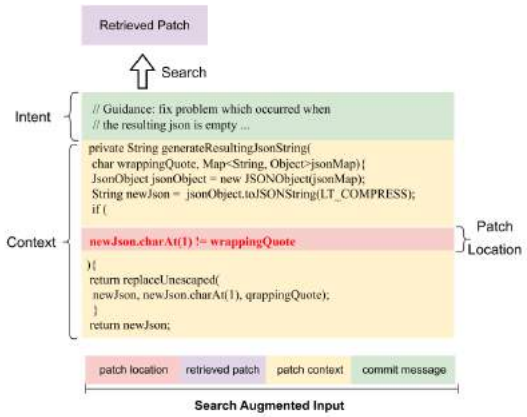



Fig. 3: Search-Augmented Input Modalities of SARGAM

techniques [37, 38, 43, 44], we assume that the developer knows the exact patch location. We realize the localization with a tree-based parser based on GumTree [45]. Patch context is the whole function body where the patch will be applied (including both the context before and after the patch location). The third modality (intent) is optional and approximated from the patch commit message. We further augment each input with a token sequence retrieved from the search step as discussed below. Each modality is separated by <s>.

The retrieved patch is inserted after the patch location. We assume that due to the small length of the patch location and retrieved patch, information from retrieved patch will not be lost during truncation. To ensure that all the models are given the same information, we use the same context window size (512 tokens after tokenization) as PLBART [14], CoditT5 [23], and NatGen [17]. Strictly following [37] and [23], for samples exceeding the context window size, we simply truncate at the end. In the fine-tuning stage, the Transformer model is trying to adaptively capture the relationship between different modalities in order to minimize the training loss. As a result, the model is supposed not to copy too many tokens when the retrieved patch or the commit message is not very similar to the expected patch. These also explain why the generation model still benefits from retrieved patches although they are not even close to the ground truth under some settings in Table V.For code editing tasks, the input of each modality is a list of tokens. We tokenize the input with the tokenizer which is compatible with the backbone generation model.

### C. Search

We maintain a database $P$ of previous patches, where each patch can be stored as tuple ($v^0$,$v^1$). In this step, given the original code version as a query, $v_q^0$, SARGAM retrieves the potential edits from the database that were applied before in a similar code context. In particular, SARGAM utilizes a brute-forced approach: it computes cosine similarities between $v_q^0$ and all the instances (an instance refers a patch along with its corresponding patch location, patch context and commit message) of $v^0$ in the database and creates a ranked list based on the similarity scores. SARGAM then retrieves $top\ k$ similar $v^0$s and fetches their corresponding patches, $v^1$s. Each retrieved patch is then augmented with the original input, as shown in Figure 3.

To ensure the information of all the modalities are passed into our system, for the retrieval model, the window size is 1024. No dimensionality reduction was performed in the search component.

**Algorithm 1:** Pseudo Code of Search

**Data:**
1. A query $v_q^0$ as an original code version to be patched,
2. Patch database $P = \{(v_1^0, v_1^1), ..(v_i^0, v_i^1), (v_N^0, v_N^1)\}$, stored with embedding of each $v_i^0 : \mathcal{E}(v_i^0)$
3. Number of patches to be retrieved $k$;

**Result:** Retrieved Patches

1 $retrievedP$ = [] ;
2 **for** $p$ *in* $P$ **do**
3 d = Distance($\mathcal{E}(v_q^0)$, $\mathcal{E}(v_p^0)$);
4 $retrievedP$.append( {patch:$v_p^1$, distance:d}) ;
5 **end**
6 Sort $retrievedP$ using distance ;
7 **return** $retrievedP$[: $k$]

Algorithm 1 shows the pseudo-code for our technique. As inputs, the algorithm takes an original code version that needs to be patched ($v_q^0$), a database of previous patches $P$, and how many patches we want to retrieve ($top\ k$). For each original version of a patch $v_p^0$ in the database, we compute its edit distance from $v_q^0$. We compute the edit distance in the embedded space to facilitate the computation. Thus, all the original code versions in the patch database are stored in its embedded form $\mathcal{E}(v^0)$, and the query is also embedded. We use PLBART [14] to get such embeddings. The edit distance is computed using cosine similarity—for any two pieces of embedded code $x$ and $y$, we compute:

$$d = 1 - \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} = 1 - \frac{\sum_{i=1}^{N} x_i y_i}{\sqrt{\sum_{i=1}^{N} x_i^2}\sqrt{\sum_{i=1}^{N} y_i^2}} \quad (1)$$

For each candidate *p* in the database, the computed distance along with the retrieved patch ($v_p^1$) is stored in a list (line 4). The final list is further sorted in descending order by distance to the query (line 6), and the algorithm returns the *top k* closest entries to the query (line 7).

Such similarity measurements simulate the situation where the developer looks for use cases on the internet and chooses the problem statement most similar to their scenario.

### D. Generation Model

Here we use three state-of-the-art edit generation models: PLBART [14], CoditT5 [23], and NatGen [17]. The output of this step is a token sequence corresponding to the generated patch. For PLBART[14] and NatGen[17], the output formats are identical to the expected patch format and no more post-processing is needed. However, CoditT5's [23] is an edit generation model; its output sequence is of the format: *Edit Operations* <s> *fixed code*. Thus, we further post-process them to create a sequence of tokens corresponding to the generated patch.

### E. Modification Model

Here, a generated code, *e.g.,* $v_{gen}$, from the previous step is further modified. We describe two basic edit operations on $v_{gen}$:

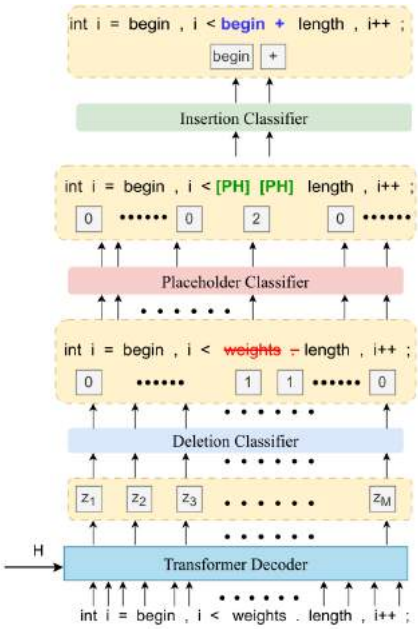



Fig. 4: Example modification steps generated by Levenshtein Transformer corresponding to the motivating example. The encoder takes patch location, context, and optional developer's intent as input and outputs hidden state $H = \{h_1, h_2, \cdots, h_N\}$, where N refers to the length of the input sequence. LevT decoder takes $H$ and patch location, and after some Transformer decoder layers, outputs $(z_1, z_2, \cdots, z_M)$. It is passed to three classifiers (deletion, placeholder, insertion) to perform the edits.

- **delete** token $d$ from $v_{gen}$.
- **insert** token $i$ at location $l$ in $v_{gen}$.

Any other code change operation, *e.g.,* replacement, move, etc., can be expressed in terms of delete and insert [12, 13]. Multiple modifications can further be expressed as a sequence of token deletion and insertion operations, resulting in the final patched code. To capture such insertion-deletion operations, we use LevT, as discussed in Section II-B. Figure 4 illustrates this step w.r.t. our motivating example (see Figure 2).

**Modeling Edits.** Given a token sequence representing $T = (t_1, t_2, ..., t_n)$, the two edit operations, deletion and insertion, are consecutively applied to generate the final output. As discussed in Section II-B, LevT decoder has three classification heads: Insertion, Deletion, and Placeholder. We model the code edit operations using these three classifiers, as follows:

*Token Deletion.* LevT reads the input sequence $T$, and for every token $t_i \in T$, the deletion classifier makes the binary decision of 1 (delete the token) or 0 (keep the token). The output of the deletion head is $T'$. Figure 4 shows that the deletion classifier identifies the tokens `weights` and `.` for deletion (marked in red).

*Token Insertion.* On $T'$, the insertion operation is realized in two phases: predicting locations to insert the token and then predicting the tokens to be inserted. First, among all the possible locations where a new token can be inserted, *i.e.,* $(t'_i, t'_{i+1}) \in T'$, the Placeholder head of LevT predicts how many placeholders can be added. Next, the insertion head of LevT replaces each placeholder with a token chosen from the vocabulary.

For instance, in Figure 4, the Placeholder Classifier predicts

two placeholder positions between tokens < and `length`, as marked by `[PH] [PH]` (*i.e.,* `i < [PH] [PH] length`). Next, the Insertion Classifier focuses only on the two placeholders and predicts `begin` and + respectively. Finally, we get the intended patch `int i = begin , i < begin + length , i++ ;`.

## IV. Experimental Design

### A. Datasets

TABLE I: Studied Code Editing & Bug-fixing Datasets

| Dataset | #Train | #Valid | #Test |
|---|---|---|---|
| **Bug2Fix small** ($B2F_s$) | 46,628 | 5,828 | 5,831 |
| **Bug2Fix medium** ($B2F_m$) | 53,324 | 6,542 | 6,538 |
| **CoCoNuT pre-2006** | 2,593,572 | 324,196 | - |
| **Defects4J**$_{1.2}$ | - | - | 75 |
| **Defects4J**$_{2.0}$ | - | - | 82 |
| **QuixBugs** | - | - | 40 |

Table I summarizes the dataset we use for our study.

*Code Editing Data:* The accuracy of the code editing task of SARGAM is evaluated by utilizing the Bug2Fix dataset [46] similar to [23, 37] (including $B2F_s$ and $B2F_m$). $B2F_s$ contains shorter methods with a maximum token length 50, and $B2F_m$ contains longer methods with up to 100 token length.

*Bug Fixing Data:* The effectiveness of the pipeline is measured with Defects4j [47] and QuixBugs[48]. The Generate and Edit parts of the pipeline are trained with CoCoNuT pre-2006 [43], which has over two million samples in the training set. Since the bugs in CocoNut pre-2006 are older than the first bug in benchmarks we used, there is no risk of having patched code in the training set. After training, we test our pipeline on (1) 75 single-line bugs in Defects4J$_{1.2}$ and (2) 85 single line bugs in Defects4J$_{2.0}$ and (3) 40 bugs in QuixBugs.

### B. Training

We trained LevT on 4 GeForce RTX3090 Ti GPUs with a 64,000 tokens batch size, following [40], and applied a dual-policy learning objective, stopping when validation set performance plateaued for 5 consecutive epochs. For code editing, we fine-tuned PLBART, CoditT5, and NatGen, using learning rates of $5e^{-5}$, with batch sizes of 16 for PLBART and 48 for CoditT5 and NatGen, adhering to strategies from relevant literature, and implemented the same early stopping criterion as in LevT training.

### C. Baselines

We fine-tune three large-scale pre-trained language generation models: PLBART [14], CoditT5 [23] and NatGen [17] on each dataset and consider them as our baselines. CoditT5[23] is an edit-generation model that generates edits in terms of token addition, deletion, and replacement operations. In contrast, NatGen [17] and PLBART [14] are code-generation models that generate a sequence of tokens. Another edit generation model, MODIT [37] studied several information modalities on top of PLBART [14]. We use MODIT's [37] recommendation to select the input modalities and report results on the different baselines. We compare SARGAM with the following deep learning-based baselines for the bug-fixing task: CocoNut [43], CURE [38], KNOD [35], and AlphaRepair [36].

### D. Evaluation Metric

We use accuracy (exact match) to evaluate the results on both code editing and program repair. When a synthesized patch is exactly identical to the expected patch, we call the synthesized patch the correct one. For code editing tasks, we report $top$ 1 and $top$ 5 accuracies. Given a retrieval augmented input, we let the code generation model output up to $top$ 5 patches; modify each of the generated patches once and produce up to 5 final candidate patches. Following [23], we apply statistical significance testing using bootstrap tests [49] with confidence level 95%. The result with the same prefixes (*e.g.,* $\alpha$ ) are not significantly different.

In the case of our program repair tool, we generate and evaluate up to the top 1250 patches. We made this choice in consideration of other APR tools, which often evaluate up to the top 5000 patches. We believe that reporting accuracy at the top 1250 is a reasonable and fair choice, particularly because our APR approach includes test cases to validate the generated patches.

## V. Results and Analysis

In this section, we empirically evaluate:

RQ1. How effective is SARGAM for code editing?

RQ2. What are the contributions of different design choices?

1) Importance of input modalities.
2) Effectiveness of a Levenshtein transformer over a vanilla transformer for patch modification.

RQ3. How effective is SARGAM for automated program repair?

### A. RQ1. SARGAM *for code editing*

*1) Motivation:* Here we investigate the core functionality of SARGAM, *i.e.,* generating code edits. We evaluate it on popular code editing benchmarks $B2F_s$ and $B2F_m$.

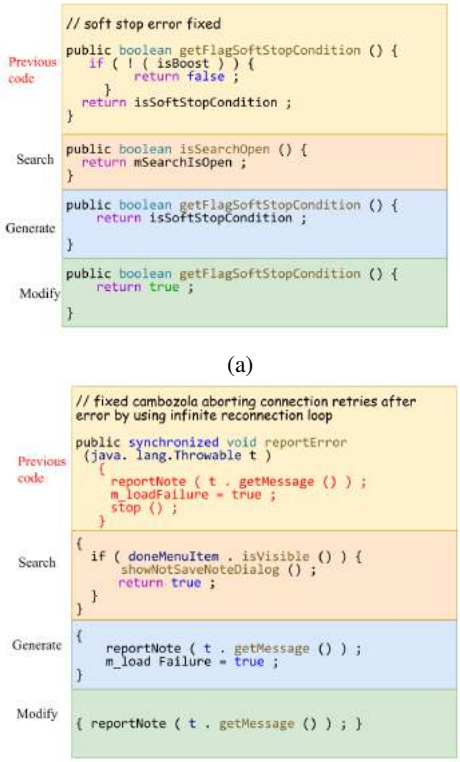
*2) Experimental Setup:* We compare SARGAM's performance with three state-of-the-art pre-trained code generation models that show effectiveness for code editing tasks: PLBART [14], CoditT5 [23], and NatGen [17]. We fine-tune all three pre-trained models on the same dataset ($B2F_s$ or $B2F_m$). While comparing with a code generation model, we incorporate the same model in SARGAM's pipeline. In that way, it shows how much SARGAM can improve compared to the corresponding generation-only setting.

In the search step, we search for similar edits from the training sets of $B2F_s$ or $B2F_m$. The retrieved patch from the training sets of $B2F_s$ or $B2F_m$ are added to the input. The generation and edit models are fine-tuned on the search augmented input. For a given retrieval augmented input, we take $top$ 1 and $top$ 5 outputs from the generation step and further modify them to produce the final patches. The reported numbers in Table II present the accuracy of the final patches.

*3) Results:* We find that SARGAM can always outperform its pure generation counterpart by a considerable margin. SARGAM can relatively improve PLBART [14], NatGen [17], and CoditT5 [23] by 19.27%, 4.82%, and 5.38% , respectively, on $B2F_s$ in terms of $top$ 1. SARGAM relatively improves

TABLE II: Exact match of SARGAM for code editing. Models in () are the off-the-shelf generative models used by SARGAM.

| Tool | $B2F_s$ | $B2F_m$ | $B2F_s$ | $B2F_m$ |
|---|---|---|---|---|
| | Top1 | | Top5 | |
| PLBART | 29.99 | 23.03 | 47.08 | 36.51 |
| SARGAM (PLABRT) | 35.77 | 27.58 | 52.43 | 37.81 |
| NatGen | 36.55 | $28.53^{\alpha}$ | 52.39 | 42.99 |
| SARGAM (NatGen) | 38.31 | $29.32^{\alpha}$ | 57.31 | **45.31** |
| CoditT5 | 37.52 | 28.33 | 54.99 | 42.32 |
| SARGAM (CoditT5) | **39.54** | **30.12** | **57.46** | 44.20 |




(a)




(b)

Fig. 5: Example correct patches generated by SARGAM. Inputs are presented in light brown boxes, and synthesized patches are presented in light green boxes.

these three backbone models by 11.36%, 9.39%, and 4.49% on $B2F_s$ in terms of $top$ 5. On $B2F_m$, SARGAM improves PLBART [14], NatGen [17] and CoditT5 [23] by 19.76%, 2.77% and 6.32% relatively in terms of $top$ 1. SARGAM also improves three backbones by 3.56%, 5.40% and 4.44% on $B2F_m$ in terms of $top$ 5.

Figure 5a shows the progress each step makes towards synthesizing the correct patch. Given the previous code as input, we retrieve a patch that is very similar to the ground truth from the code base. The Levenshtein distance between the retrieved patch and the ground truth is 2 while that between previous code and ground truth is 14. The generation model (NatGen) utilizes the retrieved patch and generates a patch based on the code context. This step brings the generated patch one step closer to the correct patch, which is only one step away from our goal. Finally, the modification model finishes the last step by deleting `isSoftStopCondition` and inserting `true`.

Figure 5b shows another example which can prove the robustness of SARGAM. "By using infinite reconnection loop" in commit message suggests that `stop()` should be wiped out from the previous code. Although the retrieved patch is not even close to the ground truth, the generation model (CoditT5 in this case) still recognizes part of the developer's intent and removes `stop()`. Based on the output of generation model, the editing model further deletes another statement `m_loadFailure = true;` and finally returns `reportNode (t.getMessage());` , which proves to be the correct patch.

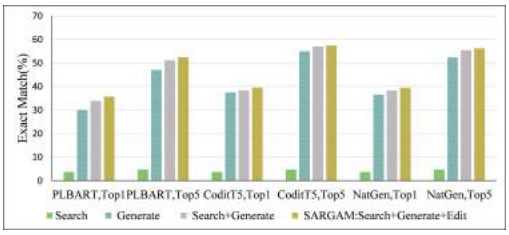



Fig. 6: Ablation Study of the steps of SARGAM on $B2F_s$

Figure 6 further shows the effectiveness of each step (search, generate, modify): for all the three off-the-shelf code generation models, adding search can improve the patch generation, and modifying the generated patch can further improve the performance. On $B2F_s$ retrieved edits can improve the $top1$ exact match of PLBART [14] by 3.97%, and the modifying step further improves it with another 1.81%. Such an improvement can also be found on $B2F_m$.

**Result 1**: SARGAM can generate better patches than generation-only or edit-only models. On average, SARGAM can improve three baseline models by 8.42% and 6.44%, in terms of $top1$ accuracy and $top5$ accuracy, respectively.

### B. RQ2. Analyzing Design Choices

*1) Motivation:* The success of the search and modification steps depends on different design choices. In particular, we investigate:

1) During Search, what is the best method to locate the most similar patch?
2) During search, which input modalities (edit location, context, and user intent) and their combinations matter for a successful patch generation?
3) How LevT outperform a vanilla Transformer-based model for patch modification?

#### *RQ2.1. Alternatives to locate the most similar patch*

*2) Experimental Setup:* Different combinations of search queries are formed using: patch location, patch context, and developer's intent. Each modality is matched with a similar modality during retrieval. TF-IDF [50], BM25 [51] and PLBART Embedding are used as baselines for patch retrieval. For PLBART Embedding and TF-IDF [50], embeddings of query $v_q^0$ and all instances in database $v^0$ are created and cosine similarity is calculated to rank and retrieve the $top$ 5 similar $v^0$ s. Additionally, BM25 algorithm is experimented with to rank and retrieve the $top$ $k$ similar $v^0$ s. Corresponding patches of the retrieved $v^0$ s are fetched. The average Levenshtein distance is computed against the ground truth.

*3) Results:* Table III shows the results. The average Levenshtein distance is computed and compared against each algorithm. Across all combinations, the best results are achieved when PLBART embedding is used for patch retrieval.

TABLE III: Impact of Different Similarity Metrics

| Patch Location | Context | Commit Message | BM25 | PLBART Embedding | TF-IDF |
|---|---|---|---|---|---|
| ✓ | - | - | 0.678 | **0.633** | 0.724 |
| - | ✓ | - | 0.719 | **0.710** | 0.726 |
| - | - | ✓ | 0.780 | **0.779** | 0.780 |
| ✓ | ✓ | - | 0.703 | **0.683** | 0.721 |
| ✓ | - | ✓ | 0.692 | **0.660** | 0.722 |
| - | ✓ | ✓ | 0.724 | **0.719** | 0.729 |
| ✓ | ✓ | ✓ | 0.704 | **0.693** | 0.719 |

*RQ2.2. Impact of Input Modalities on Search*

*4) Experimental Setup:* We form the search query with different combinations of three input types: patch location, patch context, and developer's intent. Each modality is matched with a similar modality during retrieval. We report the results both for search+generate and search+generate+modification as shown in Table IV.

TABLE IV: Impact of Different Input Modalities in Search Query on the exact match

| Patch Location | Context | Commit Message | Search+Generate $B2F_s$ | Search+Generate $B2F_m$ | Search+Generate+Modify $B2F_s$ | Search+Generate+Modify $B2F_m$ |
|---|---|---|---|---|---|---|
| - | - | - | 29.99 | 23.02 | 29.99 | 23.02 |
| - | - | ✓ | $31.97^{\alpha}$ | $23.81^{\beta}$ | $32.02^{\alpha}$ | $24.26^{\beta}$ |
| - | ✓ | - | $31.92^{\gamma}$ | 24.49 | $32.43^{\gamma}$ | 25.92 |
| - | ✓ | ✓ | **33.96** | 24.43 | 35.60 | 26.49 |
| ✓ | - | - | 31.50 | 24.72 | 32.77 | 27.58 |
| ✓ | - | ✓ | 32.82 | **25.27** | 34.01 | **27.82** |
| ✓ | ✓ | - | 31.85 | 25.01 | 33.29 | 26.11 |
| ✓ | ✓ | ✓ | 33.63 | 24.33 | **35.77** | 26.39 |

*5) Results:* Table IV shows the results of SARGAM on $B2F_s$ and $B2F_m$ with different combinations of patches retrieved as an additional modality—the retrieved patches improve the performance of the generation model, PLBART [14], across all combinations. On $B2F_s$ the best result is achieved when we use both the patches retrieved with context and that retrieved with commit message. In this case, we improve the performance of PLBART [14] by 13.24%. However, on $B2F_m$ PLBART achieves its best performance when patches retrieved with patch location and patches retrieved with commit message are passed to the input and it finally improves baseline PLBART by 9.77%.

The improvement retrieved patches bring to the generation model still holds after further modification. On $B2F_s$, using patches retrieved with all the three types of queries achieves the highest accuracy, which is actually the second best combination in the "Search+Generation" setting. On $B2F_m$, patch location & commit message is still the best combination.

Table V also reports the averaged normalized editing distance between the generated patch and the ground truth (GT).

TABLE V: Avg. Edit Distance Between the Retrieved Patch/Generated Output/Modified Output and the Ground Truth

| Patch Location | Context | Commit Message | Before Edit $B2F_s$ | Before Edit $B2F_m$ | Search $B2F_s$ | Search $B2F_m$ | +Generate $B2F_s$ | +Generate $B2F_m$ | +Modify $B2F_s$ | +Modify $B2F_m$ |
|---|---|---|---|---|---|---|---|---|---|---|
| - | - | - | 0.293 | 0.207 | - | - | - | - | - | - |
| - | - | ✓ | 0.293 | 0.207 | 0.580 | 0.759 | 0.236 | 0.196 | 0.231 | 0.191 |
| - | ✓ | - | 0.293 | 0.207 | 0.649 | 0.701 | 0.238 | 0.199 | 0.234 | 0.192 |
| - | ✓ | ✓ | 0.293 | 0.207 | 0.652 | 0.700 | **0.232** | 0.197 | **0.225** | 0.189 |
| ✓ | - | - | 0.293 | 0.207 | 0.579 | **0.608** | 0.240 | 0.195 | 0.235 | 0.190 |
| ✓ | - | ✓ | 0.293 | 0.207 | 0.580 | **0.608** | 0.235 | 0.196 | 0.231 | **0.188** |
| ✓ | ✓ | - | 0.293 | 0.207 | 0.581 | 0.609 | 0.239 | **0.193** | 0.230 | 0.192 |
| ✓ | ✓ | ✓ | 0.293 | 0.207 | **0.578** | 0.612 | 0.233 | 0.195 | 0.229 | 0.192 |

TABLE VI: Performance (exact match) of LevT and vanilla Transformer (vT) for modification

| Dataset | Gen. Models | Top1 Before Edit | Top1 vT | Top1 LevT | Top5 Before Edit | Top5 vT | Top5 LevT |
|---|---|---|---|---|---|---|---|
| small | PLBART | 33.63 | 34.85 | **35.78** | 51.15 | 51.91 | **52.43** |
| | NatGen | 38.29 | 39.23 | **39.44** | 56.58 | 57.13 | **57.31** |
| | CoditT5 | 38.38 | 39.04 | **39.55** | 57.01 | 57.37 | **57.47** |
| medium | PLBART | 25.27 | 26.32 | **27.82** | 37.57 | 37.98 | **38.82** |
| | NatGen | 27.73 | 28.47 | **29.32** | 44.23 | 44.41 | **45.31** |
| | CoditT5 | 29.29 | 30.03 | **30.12** | 43.53 | 43.68 | **44.20** |

Across all combinations, although the retrieved patch is not very similar to GT in terms of normalized editing distance, it is always helping the generation model and the modification model to synthesize patches that are closer to GT.

*RQ2.3. LevT vs Vanilla Transformer for patch modification*

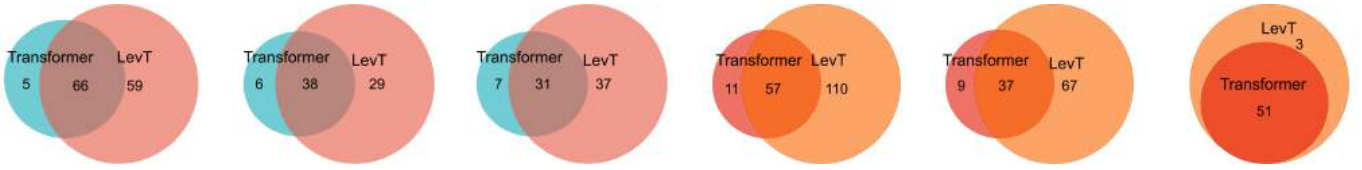
*6) Experimental Setup:* We follow the setup in V-A and use LevT and the vanilla Transformer to modify the output of generation models, which have been augmented with search results. For fairness, both LevT and the vanilla Transformer are trained on the same dataset ($B2F_s$ and $B2F_m$).

*7) Results:* Table VI reports the performance of using vanilla Transformer and LevT for editing. Across different settings, LevT always achieves a higher exact match (accuracy) of the generated edit. In addition, we present the exact numbers of overlapped and unique correct edits produced by Transformer and LevT in Figure 7. On PLBART $B2F_m$ and PLBART $B2F_s$, LevT complements Transformer by producing 110 and 59 more correct patches, respectively. Similarly, using by modifying NatGen's output, LevT can produce 29 and 67 more unique patches over vanilla Transformer for $B2F_s$ and $B2F_m$ respectively. Even when we consider CoditT5, which is an edit generation model, LevT produces 37 and 3 more unique patches over Transformer. These results show LevT is a better design choice for patch modification over vanilla Transformer.

**Result 2** The combinations of edit location, context, and developer's intent during patch retrieval can improve PLBART by up to 13.24%. LevT-based patch modification model outperforms the vanilla Transformer due to its explicit way of modeling fine granular edit operations.

## C. RQ3. SARGAM *for Bug Fixing*

*1) Motivation:* We want to check SARGAM's applicability for program repair, which is a special type of code editing task. For bug fixing, the plausibility of the generated edits can be estimated by the passing and failing test cases.




(a) PLBART $B2F_s$ (b) NatGen $B2F_s$ (c) CoditT5 $B2F_s$ (d) PLBART $B2F_m$ (e) NatGen $B2F_m$ (f) CoditT5 $B2F_m$

Fig. 7: Venn diagrams of the numbers of correct modifications made by LevT and Transformer

TABLE VII: Experiment Results (number of correct fixes) of SARGAM for Bug Fixing.

| Tool | Defect4j$_{1.2}$ | Defects4j$_{2.0}$ | QuixBugs |
|---|---|---|---|
| **CocoNut** | - | - | 13 |
| **CURE** | - | - | 26 |
| **KNOD** | 48 | 34 | 25 |
| **AlphaRepair** | 45 | 36 | 28 |
| **Codex** | 33 | 38 | 31 |
| **SARGAM** (Search+Codex+Modify) | 40 | **42** | **34** |

*2) Experimental Setup:* Following Jiang et al.'s [52] findings that Large Language Models (LLM) outperform all other DL-based repair-only models, we choose OpenAI's Codex (at zero-shot setting) [19], one of the largest code generation models at the time of writing the paper, for bug fixing. Our goal is to investigate, even using LLM, whether incorporating search and modification steps provides additional benefits.

To provide input to Codex [19], we design a prompt that combines code and natural language. Our prompt is inspired by several previous works [53–55]. The prompt consists of the following components (see Figure 8): (i) *Describing Task.* Comment at the beginning of the prompt ("fix the bug in the following method") describing Codex's task [19]; (ii) The buggy code snippet is marked with a comment "buggy line is here"; (iii) *Retrieved Patch.* The retrieved patch is augmented with comment "A possible patch for buggy line"; and (iv) *Context.* The context before the buggy line is highlighted with a comment: "change the buggy line to fix the bug:".

Here, we perform the search step in a larger training set: Java 2006. In the search step, we retrieve up to 25 similar patches, and in the generation step, we generate top50 possible patches. Hence at the inference stage, we obtain up to $(50 * 25 = 1250)$ candidate patches for every single bug. This number of candidate patches is still relatively small compared to the settings in some previous works [36, 38], which can generate up to 5,000 patches. Here, we use Defects4J test suite to validate patches after each step.

Following previous work [36], we call patches synthesized by SARGAM "candidate patches". Then we compile each candidate patch and test it against developer-written test suite to find plausible patches which can pass all the tests. Finally we check if plausible patches are exactly the same as those provided by the developer. Similar to prior work [36], we use fix correctness results collected from previous papers. For Defects4J$_{1.2}$ and Defects4J$_{2.0}$, following [38] we only use single-line bugs therefore we filter multiple-lines/hunks bugs out of the results released in the artifacts.

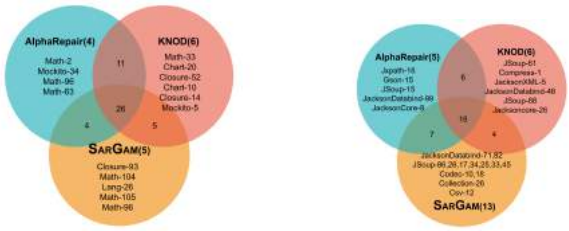
```
/// fix the bug in the following method
public void reparseCentralDirectoryData(
    boolean hasUncompressedSize,
    boolean hasCompressedSize,
    boolean hasRelativeHeaderOffset,
    boolean hasDiskStart) throws ZipException {
    if (rawCentralDirectoryData != null) {
        int expectedLength = (hasUncompressedSize
        ......
        (hasDiskStart ? WORD : 0);
        if (rawCentralDirectoryData.length !=
            expectedLength) { /// buggy line is here
    /// a possible patch for the buggy line:
    // for (int i = 0; i < (dataToWrite.length / 16); i++) {
        throw new ZipException("central
            directory zip64 extended" +
            " information extra field's length" +
            ......
        }
    }

/// Change the buggy line to fix the bug:
public void reparseCentralDirectoryData(
    boolean hasUncompressedSize,
    ......
    (hasDiskStart ? WORD : 0);

<CodeX will generate from here>
```

Fig. 8: An example prompt (Codec-17) including the buggy code (green lines), buggy line (red line), retrieved patch (yellow line), and additional context (pink lines).

*3) Results:* Table VII shows the results of SARGAM and other APR baselines on three benchmarks under the condition of perfect bug localization. SARGAM can fix more bugs than Codex [19] in all the settings showing that even if we use a really large high-capacity code generation model, search and modification steps still add values on top of it.

Overall, SARGAM fixes 42 single line bugs, outperforming all the other baselines on Defects4J$_{2.0}$, and produces 6 and 8 more correct patches than the latest APR tools AlphaRepair and KNOD, respectively. On Defects4J$_{1.2}$, SARGAM outperforms most of the deep learning baselines, but it is worse than KNOD and AlphaRepair. Note that, we report accuracies based on the top 1250 patches, whereas KNOD and AlphaRepair use 5000 patches. We believe given similar resources we will perform comparably in this setting. Table VII also presents the effectiveness of the proposed method on QuixBugs where it outperforms all the other baselines.

Figure 9 demonstrates SARGAM's unique bug-fixing capabilities alongside AlphaRepair and KNOD, with Figure 9a showing SARGAM fixing additional bugs on Defects4J$_{1.2}$ and Defects4J$_{2.0}$ —10 and 17 more than AlphaRepair, and 9 and 20 more than KNOD, respectively. Figure 9 demonstrates SARGAM's unique bug-fixing capabilities. Math-96 (Figure 10a) is a hard bug because all the `Double.doubleToRawLongBits` need to be deleted from the original sequence. Csv-12(10b) is also nontrivial because a new api method




(a) Defects4j$_{1.2}$ (b) Defects4j$_{2.0}$

Fig. 9: Unique fixes of SARGAM, AlphaRepair and KNOD.

```
if (rhs.isNaN()) {
    ret = this.isNaN();
} else {
-     ret = (Double.doubleToRawLongBits(real) ==
-             Double.doubleToRawLongBits(rhs.getReal())) &&
-         (Double.doubleToRawLongBits(imaginary) ==
-             Double.doubleToRawLongBits(rhs.getImaginary()));
+     ret = (real == rhs.real) && (imaginary == rhs.imaginary);
}
```

(a) Defects4J$_{1.2}$ Math-96

```
- public static final CSVFormat EXCEL =
-   DEFAULT.withIgnoreEmptyLines(false);
+ public static final CSVFormat EXCEL = DEFAULT
+ .withIgnoreEmptyLines(false).withAllowMissingColumnNames(true);
```

(b) Defects4J$_{2.0}$ Csv-12

```
  ...
  Document clean = Document.createShell(dirtyDocument.baseUri());
+ if (dirtyDocument.body() != null)
      copySafeNodes(dirtyDocument.body(), clean.body());

  return clean;
```

(c) Defects4J$_{2.0}$ JSoup-26

Fig. 10: Unique bugs only fixed by SARGAM

`.withAllowMissingColumnNames(true)` is called in the correct fix and it does not appear in the context. However, SARGAM is still able to fix both of them with the help of patch search and patch editing. Another example is JSoup-26 (Figure 10c), which indicates that SARGAM is able to insert a new line into the buggy code.

**Result 3**: SARGAM is capable of fixing bugs—on the real-world Java bug dataset, SARGAM can synthesize 6 and 8 more bugs than most recent AlphaRepair and KNOD.

## VI. Related Work

**Code Repair Models.** Seq2Seq models are widely explored for APR. Tufano et al. [56] applied the encoder-decoder model for bug fixing. SequenceR [28] enhanced Seq2Seq models with a copy mechanism to address the vocabulary issue. CocoNut [43] employed ensemble learning. Additionally, some research [11, 35, 57] has adopted tree/graph-based models.

LLMs like CodeBERT [58], PLBART [14], CodeT5 [16], and NatGen [17] have demonstrated significant success in APR. VRepair [59] and VulRepair [60] are T5-based model to repair vulnerabilities. Recent studies explore LLMs for zero-shot APR, eliminating the need for additional training or fine-tuning [52]. Xia et al. [36] used pre-trained CodeBERT for a cloze-style APR tool. Other works [53, 54] crafted prompts for Codex [19] for code repair as a code generation task.

The plastic surgery hypothesis [61] suggests that codebase changes often reuse existing snippets, which can be effectively identified and utilized. Despite this, many current APR methods, as derived from Neural Machine Translation, overlook leveraging the evolutionary codebase. Our approach enhances this framework by demonstrating how integrating retrieval steps with standard APR tools can unlock additional potential.

**Retrieval-based Code Repair Models.** Previous work has focused on reusing code for bug repair. Xin et. al. [62, 63] search a code database for code snippets similar to the bug context and reuses them to synthesize patches. LSRepair [64] suggests that code search accelerates the repair process, fixing some bugs in seconds. These studies inspire us to investigate whether patches from the codebase can enhance $seq2seq$ models and boost their performance. However, they depend heavily on the codebase quality; We overcome these limitations by combining code search with generation and modification models. These two components have the capability to adaptively leverage the useful information from the retrieval results and creatively synthesis patching patterns that are never seen in the existed codebase.

**Code Editing Models** Recent studies investigate DL models' ability to learn explicit edit operation [26]. Chen et al. [21] introduced pairing a graph encoder with a Transformer decoder to produce Tocopo sequences [20], representing code edits. Zhang et al. [23] developed CoditT5, a pre-trained model tailored for editing tasks. Differently, SARGAM doesn't directly produce an edit sequence but progressively steers the tool to generate the intended edit through multiple steps. [20–23] developed specialized outputs for edit operations. CoditT5 [23] generates an edit plan outlining explicit operations before producing the edited target sequence, necessitating additional post-processing. Unlike these approaches, LevT directly incorporates explicit edit operations into the decoder, bypassing the need for specialized output designs.

## VII. Threats To Validity

*External Validity.* Our edit generation evaluation relies on two datasets, $B2F_s$ and $B2F_m$, focusing on smaller edits and possibly missing broader edit characteristics. Commit messages used as edit intents can also be noisy. Despite these limitations, these datasets reflect real development practices. We also test SARGAM on three additional datasets common in APR research, ensuring our findings are robust and broadly applicable across different edit generation scenarios.

*Construct Validity.* SARGAM needs precise edit locations for input. While developers' cursors can simulate the edit location during edit generation. However, determining the exact location of a bug can be more challenging; We use other tools for bug detection, allowing SARGAM to focus on generating high-quality patches once the bug is pinpointed. This strategy emphasizes SARGAM's patch generation capabilities and avoids problems related to incorrect bug location.

*Internal Validity.* Our results may be influenced by our choices of hyperparameters used in the model (*e.g.,* learning rate, batch size, etc.). To address this concern, we released our tool as an open-source project so that other researchers and practitioners will be able to evaluate our approach in a wider range of settings, which will help to validate our findings further and minimize this potential threat.

## VIII. Discussion & Future work

**Discussion.** The primary technical innovation in our approach is the introduction of a new edit model utilizing the Levenstein Transformer. Unlike code generation, code editing incorporates the likelihood of generating edits based on a prior version of the code. To the best of our knowledge, the field of code editing models remains relatively unexplored in software engineering research, with only one previous study by Zhang et al. (2022) directly addressing edit modeling. Our empirical results demonstrate that our model surpasses the previous work, achieving improvements of 2.02% and 2.47% in Top1 and Top5 accuracy, respectively, on the $B2F_s$ dataset.

Further, we systematically capture developers' code editing behaviors within a unified framework, and empirically demonstrate: (i) Retrieval-based techniques can help a vanilla edit generation-based technique and vice versa (Table IV and Table V); (ii) A modification model can be very effective even after retrieval augmented generation; and (iii) We propose and implement a new pipeline to combine the above three steps together and prove their effectiveness.

We aim to extend our current work as follows.

**Real World Evaluation.** User study is always a crucial way to evaluate the effectiveness of tools. We plan to evaluate the utility of SarGaM through a questionnaire and ask developers about their opinion after their actual use of SarGaM.

**Limited Input Window.** The representation of the input can play an important role for Transformer models. For code editing task, we follow the input representation of [23, 37], which first concatenate all the modalities and then truncate from behind if the length of the sequence exceeds the window size. However, we may lose parts of the retrieved patch if the original patch location and its patch context are too long. Another option could be to set a window size for each of the modalities and apply truncation separately. This can ensure that all the modalities are preserved even after truncation.

**Brute-force Search**. Our patch search method employs a brute-force approach by sequentially examining code samples. This method may struggle to scale for large datasets, indicating a need for optimization towards a more efficient algorithm.

## IX. Conclusion

We propose SarGaM, a novel approach to improve pre-trained code generation models by incorporating patch search & retrieval and patch modification. Our goal is to mimic the behavior of a developer while editing, who first searches for a related patch, writes a sketch, and then modifies it accordingly. To this end, we propose a novel patch modification model based on Levenshtein Transformer, which generates fine-granular edit operations to realize patch modification. We evaluate our approach on two tasks: code editing and automated program repair. Our results demonstrate that SarGaM is highly effective for both tasks and outperforms state-of-the-art methods in most settings.

## References


[1] B. Ray, M. Nagappan, C. Bird, N. Nagappan, and T. Zimmermann, "The uniqueness of changes: Characteristics and applications," ser. MSR '15. ACM, 2015.

[2] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 180–190.

[3] M. Gharehyazie, B. Ray, and V. Filkov, "Some from here, some from there: Cross-project code reuse in github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 291–301.

[4] M. M. Rahman, J. Barson, S. Paul, J. Kayani, F. A. Lois, S. F. Quezada, C. Parnin, K. T. Stolee, and B. Ray, "Evaluating how developers use general-purpose web-search for code retrieval," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 465–475.

[5] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online q&a forum reliable? a study of api misuse on stack overflow," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 886–896.

[6] B. Ray, M. Kim, S. Person, and N. Rungta, "Detecting and characterizing semantic inconsistencies in ported code," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 367–377.

[7] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *arXiv preprint arXiv:2206.15000*, 2022.

[8] M. Boshernitsan, S. L. Graham, and M. A. Hearst, "Aligning development tools with the way programmers think about code changes," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2007, pp. 567–576.

[9] R. Robbes and M. Lanza, "Example-based program transformation," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2008, pp. 174–188.

[10] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," *arXiv preprint arXiv:1901.09102*, 2019.

[11] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural models," *IEEE Transactions on Software Engineering*, vol. 1, pp. 1–1, 2020.

[12] N. Meng, M. Kim, and K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples," *In Proceedings of 35th International Conference on Software Engineering (ICSE)*, pp. 502–511, 2013.

[13] ——, "Systematic editing: generating program transformations from an example," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 329–342, 2011.

[14] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *2021 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2021.

[15] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021. [Online]. Available: https://arxiv.org/abs/2102.04664

[16] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[17] S. Chakraborty, T. Ahmed, Y. Ding, P. T. Devanbu, and B. Ray, "Natgen: generative pre-training by "naturalizing" source code," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 18–30.

[18] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[19] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[20] D. Tarlow, S. Moitra, A. Rice, Z. Chen, P.-A. Manzagol, C. Sutton, and E. Aftandilian, "Learning to fix build errors with graph2diff neural networks," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 19–20.

[21] Z. Chen, V. J. Hellendoorn, P. Lamblin, P. Maniatis, P.-A. Manzagol, D. Tarlow, and S. Moitra, "Plur: A unifying, graph-based view of program learning, understanding, and repair," *Advances in Neural Information Processing Systems*, vol. 34, pp. 23 089–23 101, 2021.

[22] A. Connor, A. Harris, N. Cooper, and D. Poshyvanyk, "Can we automatically fix bugs by learning edit operations?" 2022.

[23] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "Coditt5: Pretraining for source code and natural language editing," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[24] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, "Jigsaw: Large language models meet program synthesis," 2022.

[25] M. Pradel and S. Chandra, "Neural software analysis," *Communications of the ACM*, vol. 65, no. 1, pp. 86–96, 2021.

[26] Y. Ding, B. Ray, P. Devanbu, and V. J. Hellendoorn, "Patching as translation: the data and the metaphor," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 275–286.

[27] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[28] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, 2019.

[29] D. Nam, B. Ray, S. Kim, X. Qu, and S. Chandra, "Predictive synthesis of api-centric code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 40–49.

[30] P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A. L. Gaunt, "Learning to represent edits," *arXiv preprint arXiv:1810.13337*, 2018.

[31] J. Zhao, F. Huang, J. Lv, Y. Duan, Z. Qin, G. Li, and G. Tian, "Do rnn and lstm have long memory?" in *International Conference on Machine Learning*. PMLR, 2020, pp. 11 365–11 375.

[32] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30*, 2017, pp. 5998–6008.

[33] G. Kobayashi, T. Kuribayashi, S. Yokoi, and K. Inui, "Attention is not only a weight: Analyzing transformers with vector norms," *arXiv preprint arXiv:2004.10102*, 2020.

[34] S. Abnar and W. Zuidema, "Quantifying attention flow in transformers," *arXiv preprint arXiv:2005.00928*, 2020.

[35] N. Jiang, T. Lutellier, Y. Lou, L. Tan, D. Goldwasser, and X. Zhang, "Knod: Domain knowledge distilled tree decoder for automated program repair," *arXiv preprint arXiv:2302.01857*, 2023.

[36] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 959–971.

[37] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 443–455.

[38] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," *arXiv preprint arXiv:2103.00073*, 2021.

[39] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," *arXiv preprint arXiv:2202.13169*, 2022.

[40] J. Gu, C. Wang, and J. Zhao, "Levenshtein transformer," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[41] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[42] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, "Jigsaw: Large language models meet program synthesis," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1219–1231.

[43] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 101–114.

[44] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 341–353.

[45] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 313–324.

[46] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 832–837.

[47] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 437–440.

[48] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, 2017, pp. 55–56.

[49] T. Berg-Kirkpatrick, D. Burkett, and D. Klein, "An empirical investigation of statistical significance in nlp," in *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, 2012, pp. 995–1005.

[50] J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242, no. 1. Citeseer, 2003, pp. 29–48.

[51] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[52] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," *ICSE*, 2023.

[53] J. A. Prenner and R. Robbes, "Automatic program repair with openai's codex: Evaluating quixbugs," *arXiv preprint arXiv:2111.03922*, 2021.

[54] H. Joshi, J. Cambronero, S. Gulwani, V. Le, I. Radicek, and G. Verbruggen, "Repair is nearly generation: Multilingual program repair with llms," *arXiv preprint arXiv:2208.11640*, 2022.

[55] Z. Fan, X. Gao, A. Roychoudhury, and S. H. Tan, "Improving automatically generated code from codex via automated program repair," *arXiv preprint arXiv:2205.10583*, 2022.

[56] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[57] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations*, 2019.

[58] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, Nov. 2020, pp. 1536–1547.

[59] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in c code," *arXiv preprint arXiv:2104.08308*, 2021.

[60] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vulrepair: a t5-based automated software vulnerability repair," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 935–947.

[61] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 306–317.

[62] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 660–670.

[63] ——, "Revisiting ssfix for better program repair," *arXiv preprint arXiv:1903.04583*, 2019.

[64] K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. Bissyandé, "Lsrepair: Live search of fix ingredients for automated program repair," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2018, pp. 658–662.