RAP-Gen: Retrieval-Augmented Patch Generation with CodeT5 for Automatic Program Repair

Weishi Wang*
weishi001@e.ntu.edu.sg
Salesforce AI Research
Nanyang Technological University
Singapore

Yue Wang*
wang.y@salesforce.com
Salesforce AI Research
Singapore

Shafiq Joty
sjoty@salesforce.com
Salesforce AI Research
USA

Steven C.H. Hoi
shoi@salesforce.com
Salesforce AI Research
Singapore

ABSTRACT

Automatic program repair (APR) is crucial to reduce manual debugging efforts for developers and improve software reliability. While conventional search-based techniques typically rely on heuristic rules or a redundancy assumption to mine fix patterns, recent years have witnessed the surge of deep learning (DL) based approaches to automate the program repair process in a data-driven manner. However, their performance is often limited by a fixed set of parameters to model the highly complex search space of APR.

To ease such burden on the parametric models, in this work, we propose a novel **R**etrieval-**A**ugmented **P**atch **Gen**eration framework (RAP-Gen) by explicitly leveraging relevant fix patterns retrieved from a codebase of previous bug-fix pairs. Specifically, we build a hybrid patch retriever to account for both lexical and semantic matching based on the raw source code in a language-agnostic manner, which does not rely on any code-specific features. In addition, we adapt a code-aware language model CodeT5 as our foundation model to facilitate both patch retrieval and generation tasks in a unified manner. We adopt a stage-wise approach where the patch retriever first retrieves a relevant external bug-fix pair to augment the buggy input for the CodeT5 patch generator, which synthesizes a ranked list of repair patch candidates. Notably, RAP-Gen is a generic APR framework that can flexibly integrate different patch retrievers and generators to repair various types of bugs.

We thoroughly evaluate RAP-Gen on three benchmarks in two programming languages, including the TFix benchmark in JavaScript, and Code Refinement and Defects4J benchmarks in Java, where the bug localization information may or may not be provided. Experimental results show that RAP-Gen significantly outperforms previous state-of-the-art (SoTA) approaches on all benchmarks, e.g., boosting the accuracy of T5-large on TFix from 49.70% to 54.15% (repairing 478 more bugs) and repairing 15 more bugs on 818 Defects4J bugs. Further analysis reveals that our patch retriever can search for relevant fix patterns to guide the APR systems.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → **Natural language processing**.

KEYWORDS

Automated program repair, Neural networks, Retrieval-augmented generation, Pretrained language models

ACM Reference Format:
Weishi Wang, Yue Wang, Shafiq Joty, and Steven C.H. Hoi. 2023. RAP-Gen: Retrieval-Augmented Patch Generation with CodeT5 for Automatic Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616256

1 INTRODUCTION

Program repair is one of the most important stages to maintain software quality, which however is a time-consuming and cost-dominating process in modern software development [49, 72]. Therefore, there have been huge needs for Automatic Program Repair (APR) tools to ease the difficulty and cost of program repair for developers with use cases including search of patches at program development time [44], build time [42, 65] or run time [9, 47].

A notable class of conventional techniques for APR is known as search-based (also referred to as generate-and-validate) approach [12, 19, 34, 51, 71, 73]. They often search for repairs based on the *fix patterns* mined via manual heuristic rules [24, 51, 60] or redundancy-based techniques [12, 19, 34–36, 73]. The latter group of approaches make a *redundancy assumption* [74] that the fixed patch can often be found (or reconstructed) from elsewhere in the codebase (a donor code snippet). This hypothesis has been validated empirically by studies [3, 43] showing that a significant proportion of commits (3%-17%) are indeed composed of existing codebase.

Meanwhile, with the recent advancement in deep learning technologies, numerous deep learning (DL)-based APR approaches [4, 7,

*Equal contribution. Corresponding author: wang.y@salesforce.com.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616256

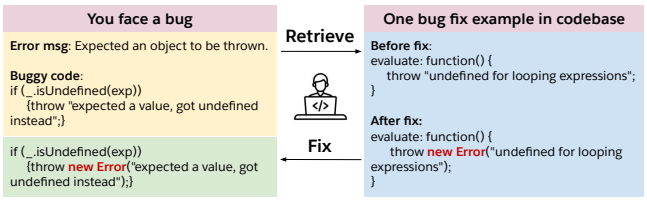

Figure 1: One motivating example of how developers fix a bug by referring to a retrieved fix pattern in codebase.

20, 40, 64, 79] have been proposed to automate the repair process via parametric models in a purely data-driven manner. In this paradigm, the APR task is typically formulated as a neural machine translation (or sequence-to-sequence learning) problem [58] in order to translate a buggy (source) program into a correct (target) version. Despite their promising results in software intelligence tasks, their performance is often limited by the fixed set of model parameters to learn the highly complex distributional patterns for program repair, even with several hundreds of million parameters [4, 20, 79].

To ease such burden on the parametric neural models, in this work, we propose a novel retrieval-augmented patch generation framework called RAP-Gen to additionally leverage relevant fix patterns from a patch retriever. Earlier APR techniques based on the redundancy assumption have shown that mining fix patterns from existing codebase [12, 19] or even external Q&As from StackOverflow [34] can serve as crucial repair ingredients for APR. Our model, which is semi-parametric in nature, aims to combine both benefits of the implicit (parametric) end-to-end program repair learning and the explicit (non-parametric) fix pattern mining. One distinction from prior fix pattern mining work is that we utilize the top relevant *bug-fix pair* as a guiding fix pattern for a buggy patch instead of clustering the fix templates with hand-crafted heuristics. This retrieval-guiding strategy is also motivated by debugging behaviours of program developers, who often search for relevant bug-fix examples to distill some repair clues for bug fixing. Fig. 1 illustrates a motivating example, where we can find that the retrieved previous repair example informs a fix pattern of wrapping the string with an "Error" object for the "throw" statement, which guides the developer to fix the target bug under consideration.

In addition, we propose to adapt a Transformer-based [68] encoder-decoder model CodeT5 [70] as the unified foundation model of RAP-Gen for both patch retrieval and generation tasks. CodeT5 is a generic code-aware language model pretrained on large source code corpora in eight popular programming languages (including JavaScript and Java) curated from GitHub, achieving state-of-the-art (SoTA) performance in both code understanding and generation tasks. RAP-Gen adopts a stage-wise learning strategy to connect the patch retriever and patch generator: the patch retriever first searches for a relevant bug fix pattern and then pass it to the CodeT5 patch generator to synthesize a ranked list of fix patch candidates based on both the source buggy code and the retrieved external bug fix knowledge. While such retrieval-augmented generation paradigm has been explored in other tasks such as question answering [23] and code generation and summarization [46], we are the first to investigate its effectiveness for APR systems based on large-scale pretrained language models for code.

For the retrievers, we propose a hybrid approach that accounts for both lexical and semantic matching through sparse (BM25 [56]) and dense (DPR [23]) retrieval based on the raw source code. We employ CodeT5's encoder as our dense DPR retriever and propose to train it with a contrastive learning objective [66] using previous bug-fix pairs as the fix patch often shares most of semantics with its buggy patch. The dense DPR retriever is expected to capture deeper code semantics while the sparse keyword-based BM25 retriever focuses more on the lexical similarity which is sensitive to the choice of naming for code identifiers. Notably, the hybrid retriever is language-agnostic as it does not require any code-specific features such as abstract syntax trees (ASTs). Experiments reveal that our patch retriever is able to retrieve lexically and semantically relevant fix patterns to guide APR systems.

We investigate the effectiveness of RAP-Gen in different APR scenarios including JavaScript linter-raised diagnostics (TFix [4]), Java bug-fix commits (Code Refinement [64]), and real Java bugs accompanied with test cases in open source projects (Defects4J [22]). Among these benchmarks, we formulate the APR problem as that given a buggy code patch, the APR model learns to predict a fix patch that repairs the bug from a codebase of previous bug-fix pairs written by developers. The correctness of the predicted fix patches are validated against either static analyzers (TFix) or unit testing (Defects4J), or via a direct comparison with the ground-truth fixes written by developers. Overall, extensive experimental results show that our RAP-Gen significantly outperforms existing DL-based methods on all these three APR benchmarks.

In summary, the paper makes the following contributions:

- We propose a novel retrieval-augmented patch generation framework (RAP-Gen) for APR. It is a generic framework that can be easily integrated with any sequence-to-sequence learning models. To the best of our knowledge, this is the first work to leverage the power of retrieval in fix pattern mining for DL-based APR systems.
- We present a hybrid patch retriever for fix pattern mining that accounts for both lexical and semantic matching through a combination of sparse and dense retrievers. It is a language-agnostic patch retriever using raw source code which does not require any code-specific features.
- We propose to adapt a generic pretrained code-aware language model CodeT5 as a foundation model for RAP-Gen to fix various bugs. Moreover, we leverage it for both patch retrieval and generation task in a unified manner.
- We extensively evaluate RAP-Gen on three APR benchmarks in JavaScript and Java. Results show RAP-Gen significantly outperforms SoTA DL-based methods on all benchmarks. Particularly, our best model yields substantial improvements (49.70 → 54.15 on exact match accuracy and 69.30 → 78.80 on error removal accuracy) on TFix over the previous SoTA T5-large model with a 3.5x larger model size than ours. On Code Refinement, RAP-Gen sets new SoTA exact match results of 24.80 and 15.84 over CodeT5's 21.61 and 13.96 for the small and medium subsets. On Defects4J, RAP-Gen achieves new SoTA performance, repairing 15 more bugs (110 → 125) with perfect FL and 6 more bugs (68 → 74) without perfect FL than the previous SoTA models.

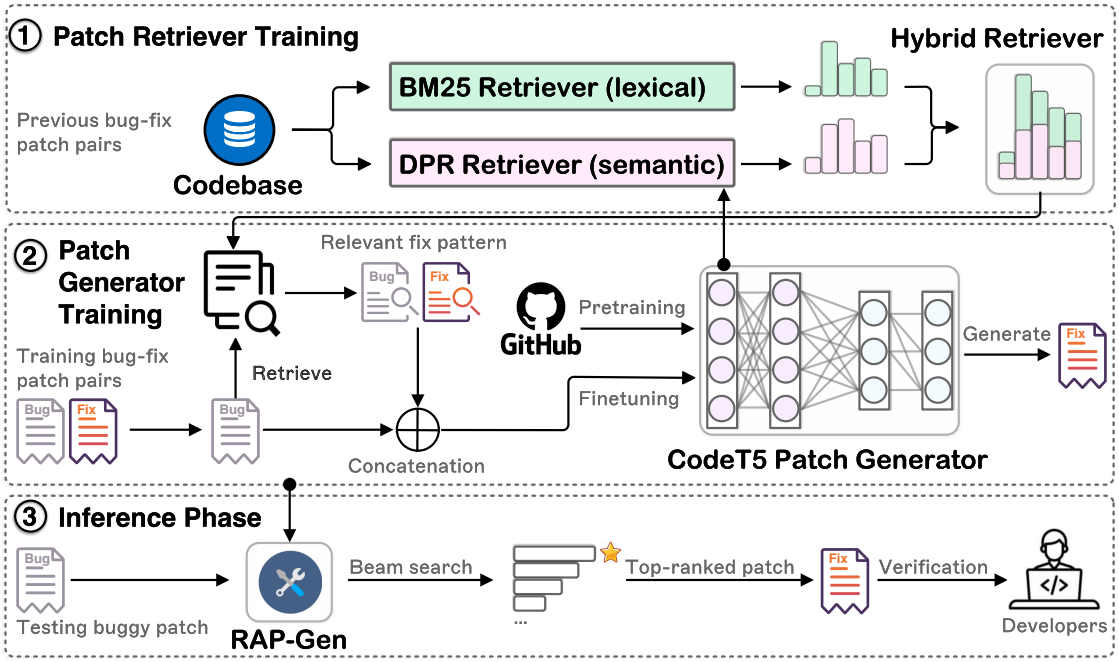

Figure 2: Retrieval-augmented patch generation (RAP-Gen) framework with CodeT5 for automatic program repair. We first retrieve the relevant bug fix patterns from the codebase through our hybrid patch retriever which takes both lexical and semantic similarity into account. We then concatenate the top-1 retrieved bug fix pattern with the query buggy patch for our patch generator to synthesize a ranked list of fix patch candidates for developers to verify.

2 RELATED WORKS

2.1 Automatic Program Repair

In the past decades, automatic program repair (APR) has attracted growing attention and various APR techniques have been proposed to reduce the manual efforts in debugging. A notable class of conventional techniques for APR is known as search-based (or generate-and-validate) approach [12, 19, 34, 51, 71, 73]. Earlier search-based APR techniques are often based on program modification or mutation with heuristic algorithm [51] or genetic programming [78] to produce a large pool of candidate fixes for validating with unit tests. The search strategy has been further extended to adopt *fix patterns* mined using redundancy-based techniques [12, 19, 26, 34–36, 73]. These approaches make a *redundancy assumption* [74] that the fixed patch can often be reconstructed from elsewhere in the codebase, which has been validated empirically by studies [3, 43] showing that a significant proportion of commits (3%-17%) are indeed composed of existing codebase. More redundancy-based techniques have shown that mining fix patterns from existing codebase [12, 19] or even external Q&As from StackOverflow [34] can largely benefit APR systems.

Recently, with the recent advancement in deep learning (DL) approaches for natural language processing (NLP), many DL-based APR techniques [4, 5, 7, 20, 40, 64, 79] have been proposed to automate the program repair process in an end-to-end data-driven manner. Motivated by the success of Neural Machine Translation (NMT), these techniques often formulate APR as a sequence-to-sequence NMT problem [58], which is to translate a buggy program into a fixed version. Various neural architectures have been explored in learning-based APR techniques. Earlier techniques [7, 64] are based on recurrent neural networks [16], which is further extended to convolution neural networks [11] in CoCoNuT [40] and Transformer-based models [68] by many recent DL-based models including TFix [4], CURE [20], Recoder [79], RewardRepair [76], and SelfAPR [75]. Notably, many of these DL-based approaches explore improving APR by leveraging code-specific features such as abstract syntax trees (ASTs) [32, 79] and test execution diagnostics [75, 76]. Specifically, Recoder [79] learns the syntax-guided edits over the ASTs to ensure the syntactic correctness of the generated fix patch, while DEAR [32] uses tree-based Long Short-Term Memory (LSTM) model [59] to better encode the code structure and constructs a suitable fixing context using surrounding AST subtrees. For the use of test execution information, SelfAPR [75] encodes test execution diagnostics into the input representation, while RewardRepair [76] improves APR with a loss function based on both program compilation and test execution information.

In terms of APR benchmarks, the most popular one would be Defects4J [22], which contains real bug-fix patches from open source GitHub projects and has been widely adopted by a large body of APR work [20, 40, 75, 76, 79]. One notable feature of this benchmark is that it contains a test suite to validate whether the bugs are fixed or not. However, as these APR approaches rely on test cases, they are inapplicable to newly discovered bugs or bugs difficult to test for deterministically [67]. Additionally, it remains a key challenge to obtain a large-scale APR dataset with test cases, e.g., one of the largest one Defects4J only contains less than 1000

bugs and another popular one QuixBugs [33] only have 40 bugs. To get rid of the requirement of test cases, there is another group of APR research [4, 41, 45, 67, 77] focusing on static analysis bugs or violations, which can be flagged by static analysis tools and is easier to curate much more bug-fix data. Besides, another type of APR [64] is based on the bug-fixing commits by checking whether the commit comments contain some keywords such as "repair" and "fix". We consider all these types of APR use cases in this work.

2.2 Pretrained Language Models for Code

Pretrained language models (LMs) like GPT [53], BERT [8], and T5 [54] have significantly boosted performance in a broad set of NLP tasks. Inspired by their success, much recent work [1, 6, 10, 14, 39, 48, 70] attempts to adapt the NLP pretraining methods to programming language. They often rely on either an encoder-only BERT-style models (CodeBERT [10] and GraphCodeBERT [14]) or decoder-only GPT-style models (CodeGPT [39] and Codex [6]), or encoder-decoder models (PLBART [1] and CodeT5 [25, 69, 70]). Particularly, CodeT5 is a unified language model pretrained with a code-aware pretraining objective on large-scale code corpora covering 8 different programming languages, which has been shown to achieve SoTA performance on a wide range of code understanding and generation tasks [39]. Compared to previous DL-based APR approaches such as CURE [20] and TFix [4] that utilize LMs pretrained primarily on natural language corpus, we propose to leverage the code-aware LMs of CodeT5 [70] for APR with better code understanding capability.

There are recent attempts [21, 50] to explore few-shot learning of large language models (LLMs) for APR. According to Prenner et al. [50], their method based on Codex achieves 46% EM compared to the finetuned T5's 59% on a random sample of 200 instances from TFix, showing that there is still a gap between few-shot learning and finetuning results. Besides, few-shot learning of LLMs requires more engineering efforts for prompting tuning and post-processing [21], which is labor-intensive. Another concern is that LLMs such as Codex [6] are not open sourced and it might be expensive to use their APIs, e.g., the Davinci version costs $0.02 for every 1K tokens[1].

2.3 Retrieval-Augmented Generation

A general retrieval-augmented generation paradigm is comprised of three components including information retrieval, data augmentation and generation model [28]. It has been widely studied in NLP and shown to achieve SoTA performance in a wide range of NLP tasks including question answering and question generation [18, 27] and machine translation [13]. Inspired by their success, much research work adapts this paradigm (also referred as retrieve-and-edit/refine framework) to benefit software intelligence tasks, including code autocompletion [15, 38], code summarization [29, 46], and code generation [46, 69].

3 APPROACH

We propose RAP-Gen, a novel retrieval-augmented patch generation framework for APR, which aims to improve APR performance by leveraging a relevant bug fix pattern retrieved from a codebase of previous bug-fix pairs. As shown in Fig. 2, our RAP-Gen framework consists of three stages: 1) a patch retriever training stage to learn a hybrid retriever that can find relevant code patches based on the lexical and semantical similarity; 2) a patch generator training stage to train a CodeT5 model to produce the fix patch based on both buggy input and retrieved bug-fix examples; 3) an inference stage to predict multiple fix patches where the top-ranked one will be passed to developers for verification.

Note that while retrieval-augmented generation techniques have been explored in many NLP tasks [18, 27], it is not trivial to adapt such techniques to APR tasks and requires systematic adaptation to address some unique challenges. The first challenge is how to retrieve relevant fix patterns for effectively guiding APR, where we build a hybrid retriever based on both lexical and semantic information, which is analyzed and compared with other retrievers in Table 8. The second challenge is how to build a top-performing APR model for various languages and APR scenarios. We leverage a language-agnostic pretrained model CodeT5 for both retrieval and patch generation, which is a more unified approach compared to prior work [38, 46] requiring a different retriever and generator.

In the remainder of this section, we first introduce the task formulation of the retrieval-augmented patch generation for APR in Section 3.1 and then revisit the backbone model of CodeT5 in Section 3.2, followed by detailing the *hybrid patch retriever* in Section 3.3 and the *retrieval-augmented patch generator* in Section 3.4.

3.1 Task Formulation

Let $\mathcal{D} = \{(X_i, Y_i)\}_{i=1}^{|\mathcal{D}|}$ be a program repair dataset consisting of $|\mathcal{D}|$ bug-fix pairs (X_i, Y_i), where X_i and Y_i are the i-th buggy and fixed program patch, respectively. Assume that we have a codebase containing a large collection of previous bug-fix pairs $\mathcal{C} = \{(B_j, F_j)\}_{j=1}^{|\mathcal{C}|}$, where (B_j, F_j) denotes the j-th previous bug-fix pair. Given a buggy program patch X_i in $\mathcal{D}$, a retriever retrieves the most relevant bug-fix pair (B_j, F_j) in the codebase $\mathcal{C}$ based on a relevance scoring function $f_\phi(X_i, B_j)$ parameterized by ϕ. Then the original input sequence X_i is augmented with the retrieved bug-fix pair to form a new input sequence $\hat{X}_i = X_i \oplus B_j \oplus F_j$, where $\oplus$ denotes the concatenation operation. The sequence-to-sequence (seq2seq) generator then generates Y_i from $\hat{X}_i$ in an autoregressive manner. Formally, we aim to learn the following probability with the patch seq2seq generator parameterized by θ:

$$P_\theta(Y_i|\hat{X}_i) = \prod_{k=1}^{n} P_\theta(Y_{i,k}|\hat{X}_i, Y_{i,1} : Y_{i,k-1})$$

where $Y_{i,1} : Y_{i,k-1}$ is the previous sequence before the k-th token and n denotes the number of tokens in the target sequence Y_i. Note that we regard the external codebase $\mathcal{C}$ as a non-parametric memory and the retrieved bug-fix pair as a guiding fix pattern for the generator. In probabilistic terms, the retrieval can be formulated as a latent variable $Z_j = (B_j, F_j)$, which is approximated by top-1 in our case. Formally, the probability can be decomposed as:

$$P(Y_i|X_i) = \sum_{j=1}^{|C|} \underbrace{P_\phi(Z_j|X_i)}_{\text{Retriever}} \underbrace{P_\theta(Y_i|X_i, Z_j)}_{\text{Generator}} \approx P_\theta(Y_i|X_i, Z_j^*)$$

where Z_j^* is the top-1 retrieved output from the retriever $P_\phi(Z_j|X_i)$. We adopt such top-1 approximation as marginalization over large k

[1] https://openai.com/api/pricing/

makes the training and inference complicated and inefficient [27]. We also tried to employ top-k ($k = 2, 3, 5$) with the Fusion-in-Decoding or FiD method [18] but did not observe a salient performance improvement.

3.2 Revisiting CodeT5

CodeT5 [70] is a unified pretrained Transformer-based encoder-decoder language model that achieves SoTA results in both code understanding and generation tasks. It is pretrained on 8.3 million functions in 8 different programming languages (i.e., Ruby, JavaScript, Go, Python, Java, PHP, C, C#) collected from GitHub. CodeT5 employs a set of identifier-aware pretraining objectives to incorporate the code-specific knowledge into the language model. In this work, we adapt CodeT5 as our dense DPR retriever and patch generator to harness its powerful code understanding capability.

BPE Subword Tokenization. One benefit of using CodeT5 is that it provides a code-specific Byte-Pair Encoding (BPE) [57] tokenizer. It can avoid the prevalent Out-of-Vocabulary (OoV) problems in the code domain as programmers tend to write arbitrary identifiers [30] and it is impossible to build a fixed vocabulary to accommodate arbitrary tokens (commonly known as *open vocabulary* problem [57]). BPE is an algorithm that learns how to efficiently split tokens into subwords based on their frequency distribution. It can also help reduce the vocabulary size as it will split rare tokens into multiple subwords instead of directly adding the whole tokens into the vocabulary. Additionally, as the CodeT5 tokenizer is pretrained and optimized for eight popular programming languages, the resulting tokenization generalizes well. As pointed out by [70], it reduces the tokenized sequence by 30% - 45% on average compared to the default T5 tokenizer [54].

Encoder and Decoder Architecture. CodeT5 consists of a stack of Transformer layers [68] for its encoder and decoder. Each Transformer layer contains a multi-head self-attention for feature aggregation followed by a feed forward layer over the output of previous layer. The final layer produces the hidden states for all input tokens, which can be employed as the code presentation for classification or generation tasks. For the CodeT5 encoder, it utilizes bidirectional attention masks to learn better contextualized representation similar to BERT [8], while the CodeT5 decoder employs causal attention masks to ensure each token can only attend to the previous tokens for better sequence generation. In RAP-Gen framework, we adapt the CodeT5 as the patch generator and its encoder specifically for the dense retriever.

3.3 Hybrid Patch Retriever

The retriever module in RAP-Gen aims to retrieve relevant fix patterns to guide the APR process. It builds on a relevance scoring function $f_\phi(X_i, B_j)$ to compute the relevance between the (query) bug X_i in $\mathcal{D}$ and a previous (key) bug B_j in the codebase C. As shown in 2 1, we utilize a hybrid approach to combine a lexical-based BM25 [56] retriever and a semantic-based DPR [23] retriever to take both lexical and semantic information into account. Prior work like [23] show that sparse and dense retriever can complement each other for more robust text retrieval.

Lexical-based Retriever. We employ BM25 [56], a well-known term-based retriever that uses sparse vector representation for lexical matching. BM25 converts each code patch as bag-of-words representation and computes a lexical similarity between the query patch X_i and a candidate patch B_j. The computed similarity score is represented as $f_\phi(X_i, B_j) = BM25(X_i, B_j)$. As a sparse term-based retriever, BM25 is sensitive to the choice of identifier naming in source code which does not impact the code semantics.

Semantic-based Retriever. We employ Dense Passage Retriever (DPR) [23] to retrieve relevant patches via measuring their semantic similarity. To encode the code patch, we use a Transformer-based encoder to map each patch to a fixed-size dense vector. Specifically, we initialize the DPR from a pretrained CodeT5 encoder and train it for a code-to-code retrieval task. For training the DPR, we propose to use the bug-fix pairs in the codebase by considering the buggy code B_j as the query and the corresponding fixed code F_j as the key. This is based on the assumption that the buggy patch and its fixed patch often shares similar semantics (e.g., identifiers and code structures). This trick avoids the massive manual annotation efforts needed to curate a bug-to-bug search dataset.

For each query patch and candidate patch, we prepend a special token of `[CLS]` into its tokenized sequence and employ the final layer hidden state of the `[CLS]` token as the patch representation. We use a shared DPR to separately encode the query patch X_i in $\mathcal{D}$ and a candidate patch B_j in C as CLS_{X_i} and CLS_{B_j}, respectively. Then the similarity is computed by the inner product between these two patch representations as the following:

$$f_\phi(X_i, B_j) = sim(X_i, B_j) = [CLS_{X_i}]^T [CLS_{B_j}]$$

For training the DPR retriever, we leverage the *in-batch negatives* to optimize an InfoNCE contrastive loss [66] defined as follows:

$$\mathcal{L}_{nce} = \frac{1}{N} \sum_{i=1}^{N} -\log \frac{\exp(sim(B_i, F_i))}{\exp(sim(B_i, F_i)) + \sum_{j \in \mathcal{M}, j \neq i} \exp(sim(B_i, F_j))}$$

where $\mathcal{M}$ is the current minibatch and N denotes the number of positive training examples in the minibatch. This objective aims to maximize the similarity between positive examples while minimizing the similarity between negative examples. Each positive example will have $|\mathcal{M}| - 1$ negative samples. Note that we do not adopt the hard negative mining strategy as in [23] due to the noisy nature of the training data.

In the inference stage, given a query buggy patch X_i, the DPR retrieves a relevant bug-fix pair (B_j, F_j) by computing the similarity between X_i (query) and B_j (key). We also tried to base on the similarity between X_i and F_j but it did not yield better results.

Hybrid Retriever. To take both lexical and semantic information into account, we utilize a hybrid approach following [23] to combine the BM25 and DPR. The similarity score is computed as $f_\phi(X_i, B_j) = sim(X_i, B_j) + \lambda\ \text{BM25}(X_i, B_j)$, where λ is a weight to balance the two retrievers and was empirically set to 1 in our experiment. Based on this combined similarity score, we select the top-1 relevant bug-fix pair (B_j, F_j) as a fix pattern to guide the patch generator for bug fixing. The hybrid retriever is expected to be more robust compared to retrievers that rely only on either lexical or semantic information.

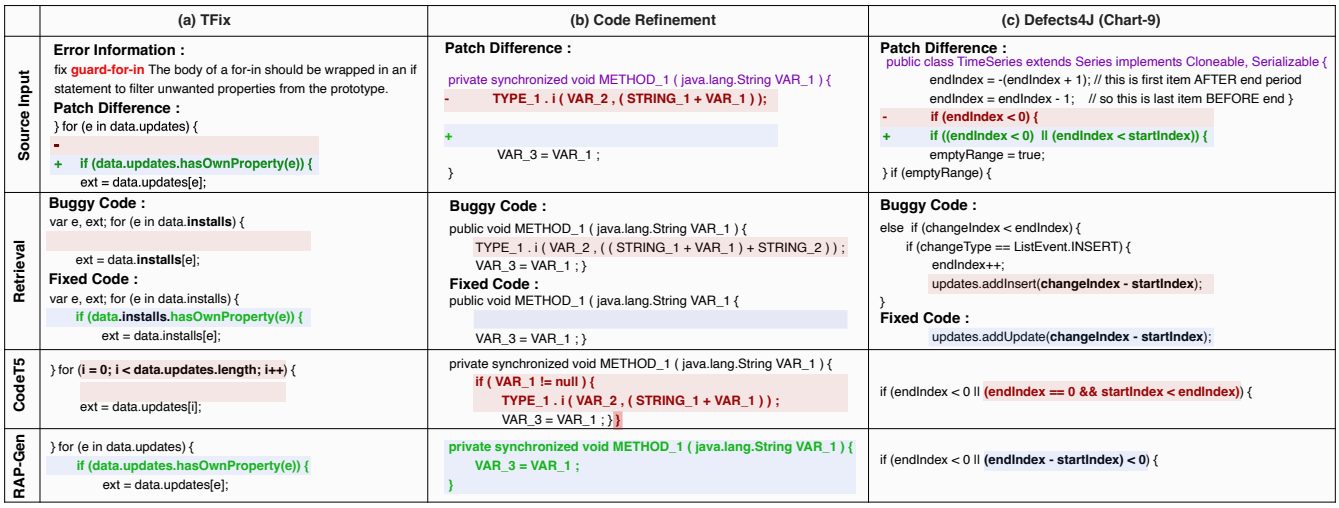

Figure 3: Bug fix examples on three APR benchmarks, where RAP-Gen successfully fix bugs while CodeT5 fails to do so.

3.4 Retrieval-Augmented Patch Generator

As shown in Fig. 2 2, given a buggy patch X_i, we search for a top relevant fix pattern (B_j, F_j) and pass it to the patch generator to generate a fixed code patch Y_i. We adopt a simple yet effective strategy to augment X_i into $\hat{X}_i = X_i \oplus B_j \oplus F_j$ via appending the retrieved bug-fix pair into the source buggy patch. Note that the patch generator module can be any sequence generation model. Different from prior studies that directly adopt a generator optimized on natural language [4], we propose to employ CodeT5, a code-aware programming language model optimized for code.

Training. We prepare the retrieval-augmented input to CodeT5 patch generator as $\hat{X}_i$ = "`[CLS]` X_i `[BUG]` B_j `[FIX]` F_j", where `[BUG]` and `[FIX]` are special tokens to separate the retrieved bug-fix pair from the buggy patch. CodeT5's encoder takes $\hat{X}_i$ as input and emits the fixed patch Y_i from its decoder in an autoregressive manner (see Section 3.1). We consider two settings of the buggy patch X_i where it may or may not contain bug localization information. If it contains error information like error type, error message, and error line, the buggy patch will be augmented to "error information `[SEP]` X_i" to incorporate error information to help fix the bugs. To train the patch generator, we adopt *teacher forcing* [63] to minimize the cross entropy loss $\mathcal{L}_{ce}$ over all training instances defined as:

$$\mathcal{L}_{ce} = -\sum_{i=1}^{|\mathcal{D}|} \log(P_\theta(Y_i|\hat{X}_i))$$

In teacher forcing, the decoder uses ground-truth context for faster convergence. We use the training set as the search codebase following [46]. To avoid information leakage, we do not allow the retriever to access the ground-truth bug-fix pair, otherwise the training loss would easily drop close to 0 as the generator can directly copy the retrieved fix as the target output. This strategy makes the training and evaluation process more compatible as the evaluation sets are not overlapped with the training set as well.

Inference with Beam Search. During inference, as shown in Fig. 2 3, we employ beam search to generate a ranked list of fixed patch candidates for an input buggy patch, where the number of predictions is determined by the beam size $\mathcal{B}$. Concretely, at each decoding timestep, the beam search selects the most $\mathcal{B}$ promising fix candidates with the highest probability using a best-first search strategy. The search process is terminated when an `[EOS]` token notifying the end of sentence is emitted. The top ranked fix patch will be examined for its correctness by comparing with ground-truth fix patches or by validating against test suites, or by manual verification by software developers.

4 EXPERIMENTAL DESIGN

4.1 Dataset

We evaluate RAP-Gen on three APR datasets, namely TFix [4] in JavaScript, Code Refinement [64] and Defects4J (v1.2) [22] in Java. All datasets are originally collected from open source GitHub commits but based on different criteria for bug identification, where TFix is based on diagnostics from a JavaScript static analyzer, Code Refinement is based on repair-related commit message, and Defects4J is based on running the test suites. We report their data statistics in Table 1.

4.1.1 TFix. TFix [4] is a large-scale program repair dataset comprising JavaScript code patch pairs curated from 5.5 million GitHub commits. It includes 52 error types (see Table 4) detected by a static analyzer ESLint[2] [62]. In addition to error types, it provides rich error annotations such as error message and localized error line so that there is no need for fault localization like prior work [20, 79]. To prepare the input sequence, as illustrated in Fig. 3(a), we follow [4] to combine all error information together with the buggy code patch into a single piece of text as the following:

fix {error type} {error message} {error context: Code Line N-1 + Buggy Line N + Code Line N+1}

[2] https://eslint.org/

Table 1: Statistics of three program repair benchmarks.

Benchmark	Version	Train	Valid	Test
TFix	Original	84,846	9,454	10,504
TFix	Deduplicated	84,673	9,423	10,465
Code Refinement	Small	46,680	5,835	5,835
Code Refinement	Medium	52,364	6,545	6,545
Defects4J	v1.2	-	-	388
Defects4J	v2.0	-	-	430

where error context consists of the given localized error line and its two neighboring code lines to form a buggy code patch. For the target sequence, it is obtained by replacing the error line into a fixed line in the error context. During data processing, we observed a *duplication issue* inside each data split and between data splits. Specifically, there are 114, 2, and 4 duplicates in the train, validation, and test split respectively, and 28, 34, and 4 duplicates for inter-split duplicates between train and test, train and test, validation and test splits respectively. We filtered all these 243 duplicates to get a deduplicated version of TFix as shown in Table 1.

Baseline Models. We compare RAP-Gen with existing DL-based APR models including SequenceR [7] and CoCoNuT [40]. Besides, we compare a large pretrained model T5-large [54] which has been finetuned on TFix to achieve the SoTA performance by [4].

Evaluation Metrics. We report **Exact Match** (EM) accuracy and **BLEU-4** score to evaluate program repair performance following [70] on TFix. BLEU-4 is a looser metric to evaluate the degree of subword overlapping while EM is a more strict metric requiring the prediction to be identical to the ground-truth patch in a real commit. As a buggy program might have different ways to repair, we further employ **Error Removal** metric following [4] to take various forms of fixes into account. The prediction is counted as correct for Error Removal if the existing error is removed and no new errors (detected by the static analyzer ESLint) is introduced after the fix. For all metrics, we present their results on a scale of 0-100 (%) and a higher score represents better performance.

4.1.2 Code Refinement. Code Refinement [64] contains bug-fix pairs at the function level, which are originally collected from public GitHub Archive[3] between March 2011 and October 2017. They use Google BigQuery APIs to identify all Java commits having a message containing the patterns: ("fix" or "solve") and ("bug" or "issue" or "problem" or "error") to ensure the quality of the collected bug-fix function pairs. They normalized the functions via obfuscating identifiers with indexed tokens such as TYPE1, VAR1, METHOD1, etc. One data example can be found in Fig. 3 (b). The dataset contains two data subsets which are determined by the number of tokens, i.e., # of code tokens ≤ 50 for the small set and $50 <$ # of code tokens ≤ 100 for the medium set. Since the bug localization is not provided, the entire code fragment is taken as the source input sequence of our model. The target sequence is the refined version of the whole code snippet.

Baseline and Metrics. We compare our RAP-Gen with pretrained programming language models based on Transformers [68]. One group of these models is the encoder-only models such as RoBERTa (code), CodeBERT [10], and GraphCodeBERT [14]. These encoder-only models require a randomly initialized decoder to generate the fix. Besides, we compare with encoder-decoder Transformer models such as PLBART [1] and CoTexT [48]. NSEdit [17] is a language model with encoder and decoder initialized from CodeBERT and CodeGPT [39] respectively. It is finetuned to generate the fix via a neural-symbolic editing sequence and ranks as the current SoTA model on Code Refinement. We follow [70] to apply BLEU-4 and Exact Match to evaluate the Code Refinement datasets.

[3] https://www.gharchive.org/

Table 2: Comparison results of CodeT5 on the original TFix.

Model	EM w/ spaces		EM w/o spaces	
	Avg.	W. Avg.	Avg.	W. Avg.
Naive Copy	0.00	0.00	0.00	0.00
SequenceR	17.90	-	-	-
CoCoNuT	11.70	-	-	-
T5-small	44.46	44.44	44.52	44.60
T5-base	48.54	47.63	48.72	47.70
T5-large	49.33	**49.65**	49.35	49.70
CodeT5-small	47.14	46.35	50.22	50.31
- error-information	45.97	45.80	49.26	49.70
CodeT5-base	**50.88**	49.42	**54.30**	**53.57**
- error-information	46.88	47.17	50.36	51.25

4.1.3 Defects4J. Defects4J [22] has been one of the most widely adopted APR benchmarks, which contains 835 real bug-fix patches in 17 open source GitHub projects. Each bug-fix example is accompanied with test cases to validate the fix. One example of Defects4J bugs can be found in Fig. 3(c), where "-" denotes a buggy line to be fixed and "+" represents the correct fix committed from a developer. A buggy line and its corresponding code context are combined to form the source input sequence while the target sequence is the fixed line. As Defects4J only has the test set, we use the project-specific training data curated by SelfAPR [75] using self-supervised learning methods. Specifically, Ye et al. [75] proposes 16 perturbation rules on the correct past version of Defects4J to construct 1,039,873 synthetic bug-fix Java patches. We use a subset of 830,240 training data that is available online.[4] For testing, we follow their exact settings to evaluate our models on 818 bugs from both Defects4J v1.2 and v2.0 (Table 1), which covers both settings with ground-truth fault localization (perfect FL) and with predicted FLs from spectrum-based FL tools such as Gzoltar [55].

Baselines and Metrics. We compare RAP-Gen with a broad set of SoTA DL-based APR models including SequenceR [7], CoCoNuT [40], CURE [20], RewardRepair [76], Recoder [79], DLFix [31], DEAR [32], BugLab [2], and SelfAPR [75]. For evaluation, we compute how many bugs can be correctly fixed on Defects4J based on unit testing and manual verification following prior work. We first run test suites to automatically identify plausible correct patches for each bug, followed by manual checking to completely verify its correctness. The correct predictions from our RAP-Gen are included in our artifact. For results of baselines, we cite the results of DLFix and DEAR from DEAR [32], and other results from SelfAPR [75].

[4] https://github.com/ASSERT-KTH/SelfAPR/tree/main/dataset

Table 3: Performance of RAP-Gen on the deduplicated TFix. Results of T5-large and CodeT5-base are different from Table 2 due to the deduplication.

Model	EM	BLEU-4
T5-large (TFix)	49.58	76.96
CodeT5-base	53.46	78.92
RAP-Gen	**54.15**	**79.66**

4.2 Implementation Details

We adopt CodeT5-base [70] that contains 12 encoder layers and 12 decoder layers with the parameter size of 220M for RAP-Gen. We implement RAP-Gen using PyTorch and train it with AdamW [37] optimizer. For the training of its neural components, we run these experiments with NVIDIA A100-40G GPUs on the Google Cloud Platform. For each benchmark, we finetune a DPR retriever for 50 epochs using the contrastive loss $\mathcal{L}_{\text{infoNCE}}$ using a batch size of 64 and a learning rate of 2e-5. We finetune RAP-Gen generator for 30 epochs using a sequence generation loss $\mathcal{L}_{ce}$ using a batch size of 32 with a learning rate of 5e-5. These best settings are obtained through a grid search for hyper-parameter tuning: batch size in (16, 32, 64) and learning rate in (1e-4, 5e-5, 2e-5). The training time of DPR retriever is 5-9 hours depending on the training size of the dataset, and the training time of RAP-Gen generator is within 2 days. For lexical-based retrievers, we use an open-sourced Python library[5] of BM25, which can be efficiently trained on CPU within one hour with multi-processing. During inference, we employ beam search with a beam size of 5 for the TFix and Code Refinement, and 100 for the Defects4J.

4.3 Research Questions

To investigate the effectiveness of RAP-Gen on APR tasks, we seek to answer the following research questions (RQs):

RQ1: Comparative study with DL-based APR models on TFix. How does RAP-Gen perform to repair JavaScript linter-flagged coding errors on TFix compared with other DL-based APR approaches?

RQ2: Analysis of RAP-Gen predictions on TFix. How does RAP-Gen repair TFix bugs for different error types and patch lengths? What fix operations do RAP-Gen adopt in repairing bugs?

RQ3: Comparative study with DL-based APR models on Code Refinement. How does RAP-Gen perform to repair Java commit-related bugs compared with other DL-based APR approaches?

RQ4: Analysis of our hybrid patch retriever. Can our hybrid patch retriever find relevant fix pattern to guide APR?

RQ5: Comparative study with DL-based APR models on Defects4J? How does RAP-Gen perform to repair Java bugs in open source projects compared with other DL-based APR approaches?

5 EXPERIMENTAL RESULT

5.1 RQ1: Comparative Study with DL-based APR Models on TFix

5.1.1 Improved TFix Evaluation. The original TFix benchmark employs the direct average of exact match (EM) accuracy across 52 error types as the main metric. However, as shown in the Table 4, these error types have a rather imbalanced distribution, e.g., the major error type "no-invalid-this" has 16,166 instances while the least error type "no-new-symbol" has only 10 instances. As such, it is more reasonable to employ the weighted average to take the error type distribution into account. Besides, we spot another limitation of its exact match evaluation that if the predicted fix contains one more whitespace such as a space or new line than the ground-truth fix, it would be regarded as a wrong exact match. However, extra whitespaces do not impact the correctness for JavaScript programs. Therefore, we propose to use the weighted average of EM w/o spaces, which normalizes the whitespaces before computing the EM to exclude the effects of the mismatch in whitespaces. As we find there is a duplication issue in the TFix dataset, we also report the results on its deduplicated version.

[5]https://pypi.org/project/rank-bm25

Table 4: Performance breakdown on 52 error types on TFix.

Error Type	#Samples	T5-large	RAP-Gen	Error Type	#Samples	T5-large	RAP-Gen
no-new-symbol	10	100.00	100.00	no-extra-bind	674	70.59	73.53
no-compare-neg-zero	13	0.00	0.00	no-case-declarations	723	58.90	67.12
no-ex-assign	40	25.00	25.00	no-fallthrough	743	76.00	77.33
for-direction	50	40.00	60.00	no-inner-declarations	830	38.10	46.43
no-unsafe-finally	63	42.86	14.29	no-array-constructor	980	86.73	85.71
use-isnan	71	37.50	25.00	no-constant-condition	1,251	48.78	54.47
no-class-assign	111	41.67	50.00	generator-star-spacing	1,396	67.86	72.86
no-dupe-class-members	117	8.33	8.33	no-extra-boolean-cast	1,458	54.11	58.22
no-func-assign	147	46.67	60.00	no-cond-assign	1,472	45.21	47.95
no-empty-pattern	178	27.78	44.44	no-process-exit	1,514	32.89	37.50
no-unused-labels	187	52.63	63.16	no-empty	2,055	26.70	31.55
no-duplicate-case	195	65.00	60.00	no-dupe-keys	2,181	53.42	55.25
getter-return	203	52.38	61.90	prefer-spread	2,466	45.08	45.49
no-sparse-arrays	237	25.00	45.83	no-useless-escape	2,920	35.15	40.61
no-const-assign	277	35.71	42.86	no-console	3,067	73.62	73.94
no-global-assign	318	59.38	68.75	guard-for-in	3,231	41.98	45.37
no-new-wrappers	360	27.78	38.89	no-throw-literal	4,075	72.06	74.51
no-this-before-super	413	47.62	69.05	no-debugger	4,164	94.48	94.24
no-unsafe-negation	423	72.09	76.74	prefer-rest-params	4,534	35.68	43.61
require-yield	429	72.09	76.74	no-unreachable	4,725	63.85	64.69
no-extend-native	443	31.11	26.67	no-extra-semi	5,969	82.61	83.61
no-new-object	446	71.11	66.67	no-redeclare	6,381	49.45	59.78
no-caller	446	20.00	22.22	comma-style	6,395	46.48	52.11
constructor-super	464	59.57	70.21	no-unused-vars	7,765	51.87	56.11
valid-typeof	539	51.85	51.85	no-undef	10,636	22.65	27.35
no-self-assign	610	34.43	44.26	no-invalid-this	16,166	37.48	44.13
				Sum/W. Avg.	104,561	49.58	**54.15**

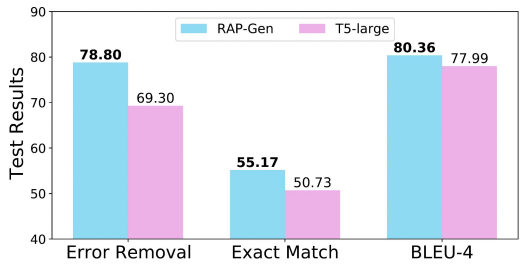

Figure 4: Error removal comparison on TFix, where error removal is well aligned with exact match and BLEU-4 scores.

5.1.2 CodeT5 Results. We compare CodeT5 models with other DL-based baselines on TFix and show results in Table 2. For the original metric of average EM w/ spaces, CodeT5-base (50.88) also yields a better accuracy than T5-large (49.33), given that it has much larger model size (~ 3.5× of CodeT5-base: 770M vs. 220M). If we focus on a more reasonable average EM w/o spaces, CodeT5-base significantly boost the performance, with around 5 absolute accuracy improvement (49.35→54.30) over T5-large. Based on the weighted average EM w/o spaces, both CodeT5-small (50.31) and CodeT5-base (53.57) outperform all the baselines including T5-large (49.70). This shows CodeT5 models with code-aware pretraining on large-scale source

Table 5: Analysis of error line removal operation on TFix.

	T5-large	CodeT5	RAP-Gen
# Ground-truth EL Removal	2,381	2,381	2,381
# Predicted EL Removal	1,882	1,925	1,922
# Correct EL Removal	1,811	1,858	1,866
# False Positive	71	67	56
Precision (%)	96.23	96.52	**97.09**
Recall (%)	76.06	78.03	**78.37**
F1 (%)	84.96	86.30	**86.73**

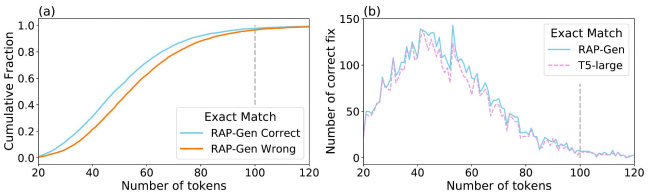

Figure 5: (a): cumulative fraction of programs by number of tokens in the source buggy patch, grouped by whether RAP-Gen can accurately fix. (b): distribution of correct fix over number of tokens for RAP-Gen and T5-large.

code have a better understanding of program. For TFix evaluation, we employ EM to denote the weighted average EM w/o spaces. We perform an ablation study to remove the error information including error type and error message from the input sequence, where we observe both CodeT5-small and CodeT5-base models have a consistent performance downgrade, revealing that it is helpful to inform which types of error they need to fix for APR models.

5.1.3 RAP-Gen Results. We report the results of our RAP-Gen model on the deduplicated TFix benchmark in Table 3, where the results are slightly different due to data size changes after duplication. Results show that RAP-Gen significantly outperforms T5-large (49.58→54.15 EM). This indicates retrieval-augmented generation is a viable and effective approach for APR and both semantic information and lexical information are crucial to retrieve relevant fix patterns. We present one case in Fig. 3 (a), where we can observe RAP-gen successfully repairs the bug with the guidance of retrieved fix pattern while CodeT5 without retrieval gives a wrong fix.

5.1.4 Error Removal Evaluation. Though exact match can ensure correctness of machine-generated patches, it might be a too strict metric to consider other forms of correct fixes. Therefore, we follow [4] to employ the error removal metric, where a fix is counted as correct if the error is removed and no new error is introduced. The error detection is based on a static analyzer ESLint. We report error removal together with EM and BLEU-4 results on a large subset of 6,793 instances[6] in Fig. 4. We observe that RAP-Gen significantly improves error removal accuracy over T5-large (69.30→78.80). The larger gain compared to EM and BLEU-4 implies that RAP-Gen is more capable of producing various forms of good fixes. Additionally, EM is well aligned with the looser metric of error removal.

[6]Some source files are unavailable to reproduce this metric on the full test set.

Table 6: Performance of RAP-Gen on the Code Refinement.

Model	Small		Medium	
	EM	BLEU-4	EM	BLEU-4
Naive Copy	0.00	78.06	0.00	90.91
LSTM	10.00	76.76	2.50	72.08
Transformer	14.70	77.21	3.70	89.25
RoBERTa (code)	15.90	77.30	4.10	90.07
CodeBERT	16.40	77.42	5.16	91.07
GraphCodeBERT	17.30	80.02	9.10	91.31
PLBART	19.21	77.02	8.98	88.50
CoTexT	21.58	77.28	13.11	88.40
NSEdit	24.04	71.06	13.87	85.72
CodeT5	21.61	77.43	13.96	87.64
RAP-Gen	**24.80**	78.28	**15.84**	90.01

5.2 RQ2: Analysis of RAP-Gen on TFix

5.2.1 Performance Breakdown on Error Types. We list the performance breakdown for 52 error types on the deduplicated TFix in Table 4. RAP-Gen outperforms the previous SoTA T5-large in 40/52 error types. Especially for the major error type "no-invalid-this", RAP-Gen improves its exact match from T5-large's 37.48 to 44.13, i.e. repairing more 107 instances. In total, RAP-Gen correctly repairs more 478 bugs than T5-large with a much smaller model size.

5.2.2 Fix Operation Analysis. We analyze what fix patterns are performed by our models on TFix. We observe a large proportion of fix consists of deletion operations compared to the code insertion and replacement operations. We find that fix operations consist of code insertion (12.5%), replacement (8.1%), deletion (47.9%), insertion and replacement (6.9%), insertion and deletion (8.2%), replacement and deletion (7.2%), and all three manners (9.2%). Earlier studies [52, 61] also reflect that the deletion operation is one of the most common fix patterns. Besides, we find one dominating fix operation is error line (EL) removal, which is to simply remove the error line from the buggy code and accounts for around 23% in the test set. We show how models perform this operation in Table 5. We observe RAP-Gen achieves the best precision, recall, and F1 scores with a lowest false positive count of 56 compared to CodeT5's 67 and T5-large's 71. This indicates that RAP-Gen is able to learn more diverse bug fix patterns instead of over relying on the trivial error line removal pattern.

5.2.3 Patch Length Analysis. We analyze the impacts of patch length Fig. 5. Fig. 5 (a) shows the cumulative fraction of buggy patches by its patch length grouped based on their outcome. We find the patches successfully repaired by RAP-Gen tend to be shorter than those where it fails. Fig. 5 (b) shows the distribution of correct fixes by its buggy patch length, where RAP-Gen can repair more bugs than T5-large especially for patches with 40 to 60 tokens.

5.3 RQ3: Comparative Study with DL-based APR Models on Code Refinement

We report the comparison results on Code Refinement in Table 6. All baseline results are directly obtained from their original papers.

Table 7: Effects of retriever modules in RAP-Gen.

Retriever	TFix	Refine-Small	Refine-Medium
No Retriever	53.46	21.61	13.96
Random	52.98	21.25	13.53
BM25	53.88	23.82	15.37
CodeBERT	52.96	22.28	15.42
CodeT5	53.93	24.37	15.60
Hybrid (BM25+CodeT5)	54.15	24.80	15.84

We first observe that "Naive Copy" gives a pretty high BLEU-4 score but with a zero exact match, indicating the buggy code and its fix has a large overlap and exact match should be employed as the primary metric. Among the baselines, NSEdit is a very competitive one with a best result (24.04 EM) on the small subset and CodeT5 gives the best result (13.96 EM) on the medium set. The lower results on the medium set compared to the small set indicates that longer buggy functions are more difficult to fix, which is aligned with observations in Fig. 5 (a). Overall, RAP-Gen achieves new SoTA results on two subsets with 24.80 EM for small set and 15.84 EM for medium set. This again confirms that retrieved fix patterns provide helpful signals to guide the program repair and the hybrid retriever is more robust by using both lexical and semantic information. Fig. 3 (b) shows one case where the retrieved fix pattern (error line removal) helps RAP-Gen to successfully fix the bug.

5.4 RQ4: Analysis of Hybrid Patch Retriever

We investigate how different retrieval modules affect the APR performance in the retrieval-augmented generation setting in Table 7. We first compare with a Random baseline via randomly retrieving bug-fix pairs from the codebase. The consistent performance downgrade compared to "no retriever" implies that randomly retrieved fix patterns cannot provide useful guiding signals for APR. Then we compare our hybrid retriever in RAP-Gen with different retrievers: sparse BM25 retrievers, and dense retrievers based on CodeBERT or CodeT5. We observe that CodeT5-based retrievers outperforms either BM25 or CodeBERT-based retrievers, while our hybrid retriever combining both BM25 and CodeT5 achieves the best APR performance, validating the effectiveness of our retriever module design in RAP-Gen.

We further analyze the performance of our retrievers in terms of lexical and semantic matching between the query and the top retrieved patches. We employ the BLEU-4 score to measure their subtoken overlap for lexical matching, while for semantic matching, we compute the cosine similarity (CosSim) between their dense vectors encoded by our fine-tuned DPR retriever. Table 8 shows the performance of our retrievers on both TFix and Code Refinement benchmarks. The first row indicates the lower-bound performance via randomly retrieving bug-fix pairs from the codebase, where we observe this Random baseline achieves much lower scores in both lexical and semantic matching.

For lexical matching, BM25 outperforms DPR (CodeT5-based) on TFix but underperforms on two Code Refinement subsets. We anticipate that it is due to the data difference between TFix and Code Refinement, where the latter employs obfuscated identifiers (e.g., VAR1, VAR2, ...) that hinders the performance of the lexical-based BM25 retriever. The hybrid retriever achieves the best lexical matching on all datasets, revealing the semantic information can complement to the lexical information. For semantic matching, DPR achieves the best results on all datasets, which is not surprising as it is optimized towards the identical objective. Notably, our hybrid retriever achieves slightly lower results than DPR but much better results than BM25, implying it can balance both lexical and semantic information and be more robust than the lexical-based retrievers, which are sensitive to the choices of identifier naming.

Table 8: Lexical (BLEU-4) and semantic (CosSim) retrieval matching results on TFix and Code Refinement benchmarks.

Retriever	TFix		Refine-Small		Refine-Medium	
	BLEU-4	CosSim	BLEU-4	CosSim	BLEU-4	CosSim
Random	0.1	35.5	14.6	35.4	14.6	30.6
BM25	23.7	70.9	41.5	68.5	39.0	66.6
DPR	21.7	**75.4**	54.4	**84.9**	44.3	**81.3**
Hybrid	**24.4**	73.4	**57.4**	84.2	**45.0**	80.9

5.5 RQ5: Comparative Study with DL-based APR Models on Defects4J

5.5.1 RAP-Gen Results. We compare RAP-Gen with other SoTA DL-based APR baselines on Defects4J [22] v1.2 and v2.0 in Table 9. We consider two settings with spectrum-based fault localization (FL) and with the perfect FL. Note that all the baseline results are cited from SelfAPR [75] and DEAR [32]. For a fair comparison, we follow common practice to adopt the same 5-hour timeout, a beam size of 100, an ensemble strategy as Recoder [79] for RAP-Gen.

As shown in Table 9, our RAP-Gen achieves new SoTA performance under perfect FL by repairing the largest set of bugs (72 bugs in v1.2 and 53 bugs in v2.0) compared to other baselines. Particularly, it repairs 7 and 8 more bugs than the previous SoTA SelfAPR in v1.2 and v2.0 respectively. For the results with spectrum-based FL, RAP-Gen achieves the second-best performance, which are very competitive to the SoTA models on both v1.2 (48 vs. Recoder's 49) and v2.0 (26 vs. SelfAPR's 28). Considering both v1.2 and v2.0 bugs, it repairs 74 bugs in total, surpassing either Recoder's 68 or SelfAPR's 67 bugs. Overall, both results with or without perfect FL validate the superiority of our RAP-Gen over other DL-based baselines. Notably, compared to many of these models, our RAP-Gen exhibits another advantage of being a language agnostic model that can generalize to other APR use cases. By contrast, Recoder requires to learn edits over AST and SelfAPR requires the test execution diagnostics, making them inapplicable or limited to deal with fragmented code snippets that cannot be parsed into ASTs or other APR scenarios without test cases.

We investigate to what extent RAP-Gen can complement existing APR models, including Recoder [79], RewardRepair [76], and SelfAPR [75]. Compared with these SoTA DL-based APR approaches, RAP-Gen repairs 13 and 12 unique bugs for Defects4J v1.2 and v2.0 respectively, which are never correctly addressed by any other DL-based APR approaches, veifying that our RAP-Gen can complement to other top-performing APR approaches. We further show a case

Table 9: Performance of RAP-Gen on Defects4J v1.2 and v2.0.

Model	Spectrum-based FL		Perfect FL	
	v1.2	v2.0	v1.2	v2.0
SequenceR [7]	-	-	14	-
BugLab [2]	-	-	17	6
DLFix [31]	30	-	40	-
CoCoNuT [40]	-	-	43	-
RewardRepair [76]	27	24	44	43
DEAR [32]	47	-	53	-
CURE [20]	-	-	55	-
Recoder [79]	**49**	19	64	-
SelfAPR [75]	39	**28**	65	45
CodeT5	27	13	58	28
RAP-Gen	48	26	**72**	**53**

In the table cells, it represents the number of correct patches. '-' indicates data unavailability.

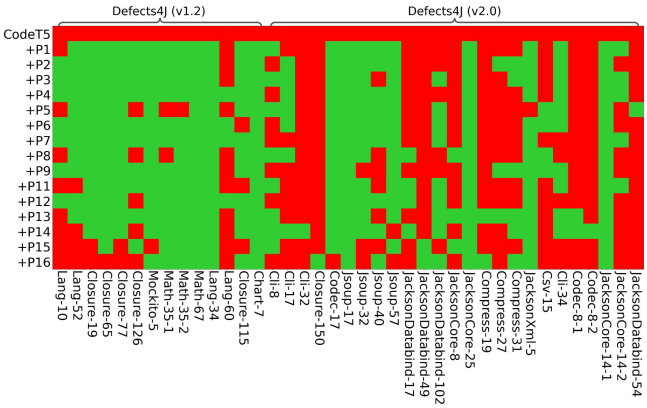

Figure 6: Effects of retrievals from various fix patterns over 39 bugs from Defects4J v1.2 and v2.0, which RAP-Gen can fix (green) and CodeT5 cannot fix (red). We represent each bug on the X-axis and use the color to denote its fixing outcome under different retrieval schemes on the Y-axis.

in Fig. 3 (c) and find that RAP-Gen successfully fixes the Chart-9 bug but in a different form with the developer's fix.

5.5.2 Effects of Retrieval from Various Fix Patterns. We analyze how retrieving a bug-fix sample from various fix patterns will affect the APR performance. For this analysis, as shown in Fig. 6, we select 39 bugs from Defects4J v1.2 and v2.0 which CodeT5 cannot fix (red) and RAP-Gen can fix (green) under the setting with perfect FL. For the categorization of fix patterns, we base on the 16 perturbation rules devised from SelfAPR [75] and use its training set for each rule as a separate retrieval codebase. We retrieve the top-1 bug-fix sample from the codebase for each rule or fix pattern (denoted as P1 to P16) and examine whether it can improve CodeT5's performance after using the guiding signals from such retrieval in RAP-Gen.

From Fig. 6, we observe that retrievals from various fix patterns in RAP-Gen are generally helpful in correcting CodeT5's predictions on Defects4J bugs. We find that most of bugs in v1.2 can be fixed after retrieval from many different patterns, while for v2.0, there are some bugs where only a few fix patterns are applicable, e.g., the P16 for Closure-150 and P5 for JacksonDatabind-54. Across various fix patterns, we find that the P13 of "insert an existing block" and P14 of "delete statement" are applicable to most bugs, indicating these are key fix patterns for repairing Defects4J bugs.

6 THREATS TO VALIDITY

Construct Validity. We evaluated RAP-Gen on three APR benchmarks: TFix in JavaScript, Code Refinement and Defects4J in Java. On TFix, we spotted a duplication issue and removed 243 intra-split or inter-split duplicates out of total 104,804 data instances. This might slightly impact the comparison between our model and the TFix (T5-large) model. We mitigate this threat by reporting the results of our model on the original TFix dataset and also the results of TFix model on the deduplicated test set. On Code Refinement, unlike the pairs in TFix can be validated by a static analyzer, its bug-fix pairs are curated from GitHub commits with a bug-fix related commit message, and only a portion of them are manually verified [64]. There is a chance that some pairs are invalid (not related to the bug fix), which brings potential threats to the reliability of the evaluation on this dataset.

Internal Validity. The threats to internal validity mainly lie in the hyper-parameter search stage for RAP-Gen. As a neural model, its performance is highly affected by the choice of hyper-parameters. To alleviate such threats, we conduct a grid search to tune a better set of hyper-parameters but we still cannot claim they are the best.

External Validity. We only evaluated our RAP-Gen model on JavaScript and Java programs and do not study its generalization to other programming languages (PLs). However, our approach is language-agnostic as we do not employ any code-specific features like ASTs and can be applied in a drop-in fashion to other PLs. Besides, our evaluation on three APR datasets in two PLs should be comprehensive enough to verify the effectiveness of our approach.

7 CONCLUSION

We present a novel retrieval-augmented patch generation (RAP-Gen) framework for automatic program repair, a fundamental task in software engineering to reduce developers' manual efforts in debugging. RAP-Gen consists of two components: a hybrid patch retriever to retrieve relevant fix patterns for a query buggy patch and a patch generator to synthesize the fixed patch based on both buggy patch and its retrieved guiding fix patterns. In addition, we propose to leverage a powerful code-aware pretrained language model CodeT5 as the backbone of RAP-Gen to facilitate both patch retrieval and generation in a unified manner. Comprehensive results on three diverse APR benchmarks in JavaScript and Java have demonstrated the effectiveness and superiority of our RAP-Gen model over existing deep learning-based APR approaches.

8 DATA AVAILABILITY

Our code and models can be found in this link (https://figshare.com/s/a4e95baee01bba14bf4b) to reproduce the results in this paper.

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *NAACL-HLT*. Association for Computational Linguistics, 2655–2668.

[2] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-Supervised Bug Detection and Repair. In *NeurIPS*. 27865–27876.

[3] Earl T. Barr, Yuriy Brun, Premkumar T. Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *SIGSOFT FSE*. ACM, 306–317.

[4] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin T. Vechev. 2021. TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer. In *ICML (Proceedings of Machine Learning Research, Vol. 139)*. PMLR, 780–791.

[5] Nghi Bui, Yue Wang, and Steven C. H. Hoi. 2022. Detect-Localize-Repair: A Unified Framework for Learning to Debug with CodeT5. In *EMNLP (Findings)*. Association for Computational Linguistics, 812–823.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021).

[7] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Trans. Software Eng.* 47, 9 (2021), 1943–1959.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT (1)*. Association for Computational Linguistics, 4171–4186.

[9] Thomas Durieux, Youssef Hamadi, and Monperrus Martin. 2017. Production-Driven Patch Generation. *2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)* (2017), 23–26.

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

[11] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. 2017. Convolutional Sequence to Sequence Learning. In *ICML (Proceedings of Machine Learning Research, Vol. 70)*. PMLR, 1243–1252.

[12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72.

[13] Jiatao Gu, Yong Wang, Kyunghyun Cho, and Victor O. K. Li. 2018. Search Engine Guided Neural Machine Translation. In *AAAI*. AAAI Press, 5133–5140.

[14] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *ICLR*. OpenReview.net.

[15] Tatsunori B. Hashimoto, Kelvin Guu, Yonatan Oren, and Percy Liang. 2018. A Retrieve-and-Edit Framework for Predicting Structured Outputs. In *NeurIPS*. 10073–10083.

[16] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[17] Yaojie Hu, Xingjian Shi, Qiang Zhou, and Lee Pike. 2022. Fix Bugs with Transformer through a Neural-Symbolic Edit Grammar. *CoRR* abs/2204.06643 (2022).

[18] Gautier Izacard and Edouard Grave. 2021. Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering. In *EACL*. Association for Computational Linguistics, 874–880.

[19] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *ISSTA*. ACM, 298–309.

[20] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *ICSE*. IEEE, 1161–1173.

[21] Harshit Joshi, José Pablo Cambronero Sánchez, Sumit Gulwani, Vu Le, Ivan Radicek, and Gust Verbruggen. 2022. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. *CoRR* abs/2208.11640 (2022).

[22] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*. ACM, 437–440.

[23] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick S. H. Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *EMNLP (1)*. Association for Computational Linguistics, 6769–6781.

[24] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *ICSE*. IEEE Computer Society, 802–811.

[25] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. In *NeurIPS*.

[26] Xuan-Bach Dinh Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *SANER*. IEEE Computer Society, 213–224.

[27] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *NeurIPS*.

[28] Huayang Li, Yixuan Su, Deng Cai, Yan Wang, and Lemao Liu. 2022. A Survey on Retrieval-Augmented Text Generation. *CoRR* abs/2202.01110 (2022).

[29] 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 *ASE*. IEEE, 155–166.

[30] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *IJCAI*. ijcai.org, 4159–4165.

[31] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: context-based code transformation learning for automated program repair. In *ICSE*. ACM, 602–614.

[32] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: A Novel Deep Learning-based Approach for Automated Program Repair. In *ICSE*. ACM, 511–523.

[33] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *SPLASH (Companion Volume)*. ACM, 55–56.

[34] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *SANER*. IEEE Computer Society, 118–129.

[35] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *ESEC/SIGSOFT FSE*. ACM, 166–178.

[36] Fan Long and Martin C. Rinard. 2016. Automatic patch generation by learning correct code. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2016).

[37] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *ICLR (Poster)*. OpenReview.net.

[38] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. *CoRR* abs/2203.07722 (2022).

[39] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021).

[40] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *ISSTA*. ACM, 101–114.

[41] Diego Marcilio, Carlo A. Furia, Rodrigo Bonifácio, and Gustavo Pinto. 2020. SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings. *J. Syst. Softw.* 168 (2020), 110671.

[42] Monperrus Martin, Simon Urli, Thomas Durieux, Matias Martinez, Benoît Baudry, and Lionel Seinturier. 2019. Repairnator patches programs automatically. *Ubiquity* 2019 (2019), 1 – 12.

[43] Matias Martinez, Westley Weimer, and Monperrus Martin. 2014. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. *Companion Proceedings of the 36th International Conference on Software Engineering* (2014).

[44] Kivanç Muslu, Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2012. Speculative analysis of integrated development environment recommendations. In *OOPSLA*. ACM, 669–682.

[45] Wonseok Oh and Hakjoo Oh. 2022. PyTER: effective program repair for Python type errors. In *ESEC/SIGSOFT FSE*. ACM, 922–934.

[46] Md. Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval Augmented Code Generation and Summarization. In *EMNLP (Findings)*. Association for Computational Linguistics, 2719–2734.

[47] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, W. Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. 2009. Automatically patching errors in deployed software. In *SOSP '09*.

[48] Long N. Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James T. Anibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: Multi-task Learning with Code-Text Transformer. *CoRR* abs/2105.08645 (2021).

[49] Strategic Planning. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology* (2002), 1.

[50] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's Codex Fix Bugs?: An evaluation on QuixBugs. In *APR@ICSE*. IEEE, 69–75.
[51] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *ICSE*. ACM, 254–265.
[52] Zichao Qi, Fan Long, Sara Achour, and Martin C. Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA*. ACM, 24–36.
[53] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).
[54] 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. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67.
[55] André Riboira and Rui Abreu. 2010. The GZoltar Project: A Graphical Debugger Interface. In *TAIC PART (Lecture Notes in Computer Science, Vol. 6303)*. Springer, 215–218.
[56] Stephen E. Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 3, 4 (2009), 333–389.
[57] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *ACL (1)*. The Association for Computer Linguistics.
[58] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).
[59] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *ACL (1)*. The Association for Computer Linguistics, 1556–1566.
[60] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated repair of software regressions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 471–482.
[61] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *SIGSOFT FSE*. ACM, 727–738.
[62] Kristín Fjóla Tómasdóttir, Mauricio Finavaro Aniche, and Arie van Deursen. 2017. Why and how JavaScript developers use linters. In *ASE*. IEEE Computer Society, 578–589.
[63] Nikzad Benny Toomarian and Jacob Barhen. 1992. Learning a trajectory using adjoint functions and teacher forcing. *Neural Networks* 5, 3 (1992), 473–484.
[64] 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 Trans. Softw. Eng. Methodol.* 28, 4 (2019), 19:1–19:29.
[65] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Monperrus Martin. 2017. How to Design a Program Repair Bot? Insights from the Repairnator Project. *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)* (2017), 95–104.
[66] Aäron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation Learning with Contrastive Predictive Coding. *CoRR* abs/1807.03748 (2018).
[67] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *ICSE*. ACM, 151–162.
[68] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS*. 5998–6008.
[69] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *arXiv preprint* (2023).
[70] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *EMNLP (1)*. Association for Computational Linguistics, 8696–8708.
[71] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. *2009 IEEE 31st International Conference on Software Engineering* (2009), 364–374.
[72] Cathrin Weiß, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. 2007. How Long Will It Take to Fix This Bug?. In *MSR*. IEEE Computer Society, 1.
[73] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *ICSE*. ACM, 1–11.
[74] Martin White, Michele Tufano, Matias Martinez, Monperrus Martin, and Denys Poshyvanyk. 2019. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2019), 479–490.
[75] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *ASE*. ACM, 92:1–92:13.
[76] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-based Backpropagation. In *ICSE*. ACM, 1506–1518.
[77] Hiroaki Yoshida, Rohan Bavishi, Keisuke Hotta, Yusuke Nemoto, Mukul R. Prasad, and Shinji Kikuchi. 2020. Phoenix: a tool for automated data-driven synthesis of repairs for static analysis violations. In *ICSE (Companion Volume)*. ACM, 53–56.
[78] Yuan Yuan and Wolfgang Banzhaf. 2020. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Trans. Software Eng.* 46, 10 (2020), 1040–1067.
[79] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *ESEC/SIGSOFT FSE*. ACM, 341–353.

Received 2023-02-02; accepted 2023-07-27