Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair

Yuxiang Wei
University of Illinois
Urbana-Champaign, USA
ywei40@illinois.edu

Chunqiu Steven Xia
University of Illinois
Urbana-Champaign, USA
chunqiu2@illinois.edu

Lingming Zhang
University of Illinois
Urbana-Champaign, USA
lingming@illinois.edu

ABSTRACT

During Automated Program Repair (APR), it can be challenging to synthesize correct patches for real-world systems in general-purpose programming languages. Recent Large Language Models (LLMs) have been shown to be helpful "copilots" in assisting developers with various coding tasks, and have also been directly applied for patch synthesis. However, most LLMs treat programs as sequences of tokens, meaning that they are ignorant of the underlying semantics constraints of the target programming language. This results in plenty of statically invalid generated patches, impeding the practicality of the technique. Therefore, we propose Repilot, a general code generation framework to further copilot the AI "copilots" (i.e., LLMs) by synthesizing more *valid* patches during the repair process. Our key insight is that many LLMs produce outputs autoregressively (i.e., token by token), resembling human writing programs, which can be significantly boosted and guided through a Completion Engine. Repilot synergistically synthesizes a candidate patch through the interaction between an LLM and a Completion Engine, which 1) prunes away infeasible tokens suggested by the LLM and 2) proactively completes the token based on the suggestions provided by the Completion Engine. Our evaluation on a subset of the widely-used Defects4j 1.2 and 2.0 datasets shows that Repilot outperforms state-of-the-art techniques by fixing 27% and 47% more bugs, respectively. Moreover, Repilot produces more valid and correct patches than the base LLM with the same budget. While we focus on leveraging Repilot for APR in this work, the overall approach is also generalizable to other code generation tasks.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Automatic programming*.

KEYWORDS

Program Repair, Large Language Model, Completion Engine

ACM Reference Format:
Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated 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/10.1145/3611643.3616271

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/10.1145/3611643.3616271

1 INTRODUCTION

Automated Program Repair (APR) seeks to reduce the manual bug-fixing effort of developers by automatically synthesizing patches given the original buggy code [20]. State-of-the-art traditional APR tools are mainly based on handcrafted repair templates to match the buggy code patterns and apply the corresponding code changes [21, 41]. Although outperforming other traditional techniques [37, 43, 47], such tools can only fix the bug types within the preset templates and cannot generalize to new bug types. With the development of Deep Learning (DL) techniques, researchers build learning-based APR [29, 72, 74] tools based on Neural Machine Translation (NMT) [57] architecture. They train NMT models to translate buggy code into correct code by learning from pairs of buggy and fixed code scraped from open-source commits. However, as discussed in prior work [67], the training sets of these tools can be limited in size and also contain irrelevant or noisy commits.

More recently, researchers have leveraged the growth in the field of NLP to directly use Large Language Models (LLMs) [10, 17] for APR [31, 66, 67]. LLMs not only achieve impressive performance on many NLP tasks [7], but are also shown to be reliable "copilots"[1] in assisting developers with various coding tasks [4, 40]. The reason is that modern LLMs often include large amounts of available open-source code repositories as part of their training dataset. Recognizing the power of LLMs, researchers have recently applied LLMs for APR: instead of translating buggy code into correct code, LLMs are directly used to synthesize the correct patch from the surrounding context. AlphaRepair [67] reformulates the APR problem as a *cloze* (or infilling) task [2, 19]: it first replaces the buggy code snippets with masked tokens and then uses CodeBERT [17] to fill correct code in given the surrounding context. Other studies on LLMs for APR have applied even larger LLMs with different repair settings (including generating complete patch functions) [33, 53, 66].

While prior LLM for APR techniques achieve state-of-the-art bug-fixing performance, they use LLMs in a black-box manner, where the underlying LLM generate programs according to the token distribution without any structural or semantic understanding of the code. To highlight the limitations with current LLMs for APR tools, In Figure 1 we show 3 scenarios where LLM can generate incorrect patches. ❶ **Generating infeasible tokens**. In Figure 1.1, the LLM has a high probability (>90%) of generating `String` to complete the `asString` method. However `asString` is not a valid field access for the object `t` and is also not part of the scope of the current

[1] One popular AI pair programmer tool (based on Codex [10]) is named Copilot [22].

Figure 1: Limitations of existing LLM-based APR approaches.

buggy method. In this case, the patchs generated using `asString` will never be correct as it cannot compile. By directly using the model probabilities, LLMs are likely to generate many patches using invalid tokens and decrease the likelihood of generating the correct patch with `End` (0.2%). ❷ **Hard to generate rare tokens**. LLMs usually cannot generate a complete identifier name in one step since it uses subword tokenization [55] to break uncommon words into smaller subwords. These uncommon words manifest as rare identifiers in code, where identifier names are CamelCase or underscore combinations of multiple words (e.g., `asEndTag` in Figure 1.2). As such, LLMs need to generate these identifiers step by step, needing not only multiple iterations but also accurate output in each step. Since prior approaches [33, 66] sample based on probability, the likelihood of completing a rare token to fix a bug can be extremely low. ❸ **No explicit consideration of types**. In addition to potentially generating out-of-scope identifiers, LLMs do not have access to various type information that gives hints to the valid identifiers. In Figure 1.3, the return type of `asEndTag()` is `EndTag`, whose definition is not explicitly given to the LLM in its immediate context. As such, LLMs do not know the correct member fields of `EndTag` and may generate invalid patches containing identifiers that do not fit the required type. On the contrary, a Completion Engine has full access to the project and can easily figure out the return type of `asEndTag()` through static analysis on the abstract syntax tree of the program. By treating code as a sequence of textual tokens, the important type information is not encoded.

To address the aforementioned limitations, we propose Repilot, a framework to further copilot the AI "copilots" (i.e., LLMs) via fusing LLMs with Completion Engines to synthesize more *valid* patches. Completion Engines [48] can parse incomplete programs and reason about the semantics in an error-tolerant manner. Our key insight is *to liken LLM autoregressive token generation as a human developer code writing, where the Completion Engine can provide real-time updates to check if the human/LLMs written partial code is valid*. Repilot first uses the LLM to provide the probabilities of generating the next token in the patch and then queries the Completion Engine to modify the probability list by dynamically zeroing the probabilities of invalid tokens. We can then sample from the new probability list to select the next token. Furthermore, recognizing the ability for Completion Engines to suggest completions, we use this feature whenever there is only one possible identifier suffix to complete the context. This not only allows Repilot to generate patches with valid rare and long identifiers but also reduces the work of LLMs needed to iteratively generate long identifier names.

For example, Repilot directly prunes the `String` and `Name` tokens in Figure 1.1 as they are infeasible according to the Completion Engine, but still accepts the correct `End` token. In Figure 1.2, the Completion Engine recognizes that `asEndTag` is the only valid continuation to the prefix `asEnd`, so Repilot directly completes this token without querying the LLM. To combat the time cost of Completion Engine, we implement several optimization techniques to minimize the overhead. Note that the recent SYNCHROMESH work [52] also employs a Completion Engine for reliable code generation with LLMs. However, it relies on expert-designed constraints and only targets domain-specific languages (e.g., SQL). Repilot directly works for general-purpose programming languages while introducing minimal overhead and can proactively complete the current generation using the Completion Engine without querying the LLM.

To demonstrate the generalizability of Repilot, we instantiate Repilot with two LLMs having distinct architectures and sizes: CodeT5-large [61], an encoder-decoder LLM with 770 million parameters, and INCODER-6.7B [19], a decoder only LLM with 6.7 billion parameters, both capable of code infilling from prefix and suffix context. We further implement a Java Completion Engine for Repilot based on the Eclipse JDT Language Server [1, 18] since it provides various semantics-based analyses through a consistent Language Server Protocol [48]. We evaluate Repilot on a subset of the widely studied Defects4J 1.2 and 2.0 datasets [32] and demonstrate state-of-the-art results in both the number of correct fixes and compilation rate — the percentage of the generated patches that can be successfully compiled. Furthermore, while we evaluated Repilot for APR in this work, we believe the overall framework can be easily applied to other code generation tasks, including code completion [16, 73], program synthesis [40, 52], and test generation [14, 65]. In summary, we make the following contributions:

- **Direction.** We open a new direction for fusing LLMs with Completion Engines for more powerful APR and beyond. Compared to prior techniques which either perform post-processing to fix invalid generations or use simple static methods to approximate these valid tokens, our approach leverages a powerful Completion Engine to directly provide accurate feedback on partial programs to avoid invalid token generations.
- **Technique.** We implement Repilot, an LLM for APR approach instantiated with the CodeT5 and INCODER models to perform cloze-style repair combined with our modified Eclipse JDT Language Server [1, 18] as the Completion Engine. In Repilot, we use the Completion Engine to systematically prune invalid tokens generated by LLMs and to directly complete code given the current prefix. Furthermore, we implement optimizations to significantly reduce the overhead of Repilot. We have open-sourced our tool at: https://github.com/ise-uiuc/Repilot.
- **Study.** We compare Repilot against state-of-the-art APR tools on Defects4J 1.2 and 2.0. Repilot is able to achieve new state-of-the-art results of 66 Defects4J 1.2 single-hunk bugs and 50 Defects4J

2.0 single-line bugs fixed respectively with 30 more combined fixes across both datasets compared to the previous best baseline. Our further evaluation shows that Repilot consistently improves the validity and correctness of the generated patches with a limited overhead (7% for CodeT5 and negligible for InCoder).

2 BACKGROUND AND RELATED WORK

2.1 Large Language Models for Code

Recent advances in Natural Language Processing (NLP) have empowered the idea of using Large Language Models (LLMs) that are pre-trained on enormous corpora of natural language and code for various code-related tasks [4, 5, 10, 38, 70]. LLMs are based on the transformer architecture [59] that can be categorized into **encoder-only**, **decoder-only** and **encoder-decoder**. Encoder-only models use only the encoder component by training using Masked Language Modeling (MLM) [15] objective where a small percentage (e.g., 15%) of the tokens are masked on. The goal of MLM is to recover these masked tokens given the surrounding context. Encoder-only models such as CodeBERT [17] and GraphCodeBERT [23] are designed to provide a representation of the input code to be used for downstream tasks such as code classification [71]. Decoder-only models, on the other hand, aim to autoregressively generate tokens based on all previously generated tokens. CodeGEN [50, 51], Codex [10] and PolyCoder [70] are examples of decoder-only LLMs where they can be used for code autocompletion tasks. Different from encoder- and decoder-only LLMs, encoder-decoder models (e.g., CodeT5 [60, 61] and PLBART [3]) combine both encoder and decoder together and jointly train both components together. A commonly used pre-training objective for encoder-decoder models is Masked Span Prediction (MSP) where random spans (multiple consecutive tokens) are replaced with single masked tokens and the models learn to *fill in* the masked span with the correct sequence of tokens. Furthermore, decoder-only models like InCoder [19] can also do infilling through the causal language modeling [2] objective. Instead of using the decoder to predict the next token in the original training data, similar to MSP, InCoder also replaces random spans with masked span tokens. During training, InCoder learns to autoregressively recover the original spans. With this training strategy, InCoder can perform infilling with bidirectional context similar to encoder-decoder models, enabling cloze-style repair.

2.2 Code Completion

Code completion is one of the most frequently used features in Integrated Development Environments (IDEs). It substantially alleviates the complexity of software development by interactively suggesting program constructs after the user's caret position while programmers are typing, including identifier names and library APIs. Code completion is now an indispensable infrastructure of the most widely-used programming languages and can be easily integrated into most modern text editors thanks to the presence of the Language Server Protocol [48], which standardizes the communication between tools and language services. Traditionally, a *semantics-based Completion Engine* is implemented on top of a series of complex *incremental* syntactic and semantic analyses of the target programming language, since it needs to understand partially written programs and provide real-time feedback. The Completion Engine has full access to a project repository and its dependencies and can produce suggestions according to its semantic understanding. Recent advances in LLMs demonstrate the capability of generating long and complicated completions. However, they may produce unreasonable programs due to the limitation in the code context size and the loss of program analysis by simply treating programs as token sequences. In this paper, we use the term Completion Engine to refer to the *semantics-based* one. We formally define the expected properties of a Completion Engine in our framework in Definition 3.4.

2.3 Automated Program Repair

Automated Program Repair (APR) aims to generate patches given the buggy code location and the bug-exposing tests. Traditionally, APR approach can be categorized as constraint-based [13, 35, 43, 47], heuristic-based [36, 37, 63] and template-based [21, 25, 34, 41, 42, 46]. Among these classic techniques, template-based tools have been shown to achieve the highest number of bug fixes by using handcrafted repair templates to target specific bug patterns [21]. However, these handcrafted patterns cannot cover all types of bugs that exist and as such, template-based tools cannot fix bugs outside of their pre-determined templates.

To address the issue faced by template-based APR tools, researchers resort to Neural Machine Translation (NMT) [57] to develop NMT-based APR tools [11, 29, 39, 44, 72, 74]. NMT-based APR tools train an NMT model to translate the input buggy code into the correct code through bug-fixing datasets containing pairs of buggy and fixed code. However, these bug-fixing datasets may contain only a small number/types of bug fixes, especially compared to a large amount of available open-source code snippets, due to the difficulty in obtaining bug-fixing commits [67]. Additionally, the datasets can fail to filter out unrelated commits [30] such as refactoring, which adds noise to the training datasets. Due to this reliance on training using bug-fixing datasets, these NMT-based tools also cannot generalize to bug types not seen during training.

Recently, researchers begin to directly apply LLMs for APR [66]. AlphaRepair [67] is the first to directly use LLMs for *cloze-style* (or infilling-style) APR: it masks out the buggy code snippet and then uses CodeBERT [17] to directly fill in the correct code given the surrounding context. While AlphaRepair demonstrates the potential to use encoder-only models for cloze-style APR, other studies [33, 53, 66] have looked into applying all three types of LLM architecture. FitRepair [64] further improves AlphaRepair via domain-specific fine-tuning and prompting strategies leveraging the plastic surgery hypothesis [6]. Even more recently, researchers have applied dialogue-based models for APR [8, 56, 68, 69]. For example, ChatRepair [69] proposes a fully automated conversational APR approach by learning from prior patching attempts, including both patch code and test failure information.

Compared to traditional and NMT-based APR techniques, LLM-based techniques are able to achieve new state-of-the-art bug-fixing results [66, 67]. While the performance is impressive, one particular limitation of these techniques is the lack of guidance in patch generation. Prior work mainly treats the LLM as a black box and only queries the model via beam search [67] or sampling [33, 53, 66]. This

means LLMs, while powerful, may still generate invalid patches given the current code context.

In this work, we address these limitations by using a *semantics-based* Completion Engine to guide and prune the LLM search space. Our approach is orthogonal to recent LLM-based APR techniques and can be easily combined with them. In fact, NMT-based APR techniques have also attempted to tackle this problem. CURE [29] first statically obtains the valid identifiers and forces the NMT model to only select from valid identifiers during generation. Recoder [74] builds an edit-based NMT model to enforce syntax correctness and introduce placeholder tokens and then as a post-processing step, Recoder will replace placeholder tokens with statically determined valid identifiers. RewardRepair [72] on the other hand, attempts to increase the number of compilable patches by penalizing uncompilable patches during training. Compared to these prior techniques, Repilot is more general and effective. Repilot does not require any domain-specific training and leverages the incremental analysis of off-the-shelf Completion Engines to enforce guaranteed constraints to guide LLMs on the fly.

3 PRELIMINARIES

In this section, we first define concepts about programming languages used throughout the paper (§3.1). Then we discuss the *formal abstractions* of the two key components used in our Repilot framework: Completion Engine (§3.2) and Large Language Model (§3.3). These two abstractions are crucial in that each of them describes a collection of fitting *implementations*, which forms the reason why Repilot is a generalizable framework.

3.1 Languages with Static Checking

We now introduce the concept of programming languages equipped with static checking and define the feasibility of a partial program before the formulation of the Completion Engine (Definition 3.3).

Definition 3.1 (Programming Language with Static Checking). A programming language with static checking is defined as a pair of its character set Σ_{PL} and its static specification $\Phi \subseteq \Sigma_{\text{PL}}^*$ as a unary relation on Σ_{PL}^*.

$$\mathbf{PL}_{\text{S}} = (\Sigma_{\text{PL}}, \Phi), \tag{3.1}$$

Given a $prog \in \Sigma_{\text{PL}}^*$, the notation $\Phi(prog)$ (or $prog \in \Phi$) states that *prog* is a statically valid program in this language. For statically-typed programming languages like Java, the compilation check is a kind of static checking.

Definition 3.2 (Static Feasibility of A Partial Program). For a partially written program $prog \in \Sigma_{\text{PL}}^*$, we say it is feasible at the caret position *caret* with respect to the static specification Φ, written as $(prog, caret) \vDash \Phi$, if and only if there exists a possible continuation after *caret* with which completing *prog* results in a statically valid program. The definition can be formally written as

$$(prog, caret) \vDash \Phi \triangleq \exists cont \in \Sigma_{\text{PL}}^*, \Phi(prog\,[caret \leftarrow cont]), \tag{3.2}$$

where we use the notation $prog\,[caret \leftarrow cont]$ as the action of completing *prog* at *caret* with *cont*, i.e.

$$prog\,[caret \leftarrow cont] \triangleq prog_{0..caret} \cdot cont \cdot prog_{caret..|prog|}. \tag{3.3}$$

In Algorithm 1, we extend this notation to accept a *range*: $\mathbb{N} \times \mathbb{N}$, so that $prog\,[range \leftarrow hunk]$ specifies the action of replacing *prog*'s contents within *range* with *hunk*.

3.2 Abstraction of Completion Engines

A Completion Engine, showed in Figure 2, provides suggested continuations to a partially written program given the caret position.

Definition 3.3 (Completion Engine). Formally speaking, a Completion Engine **CE** is a pair

$$\mathbf{CE} = (\Sigma_{\text{PL}}, complete), \tag{3.4}$$

where Σ_{PL} is the character set of the target language, and

$$complete\colon (\Sigma_{\text{PL}}^*, \mathbb{N}) \to \mathcal{P}(\Sigma_{\text{PL}}^*) \cup \{\textbf{unknown}\} \tag{3.5}$$

is a function to obtain the completions given a program at some caret position, with **unknown** indicating the engine cannot determine the suggestions from the code context (e.g., when completing a variable declaration). Note that we make a distinction between **unknown** and empty completions $\varnothing$ because in this paper we are interested in a specific group of *strict* Completion Engines that helps determine the feasibility of a partial program.

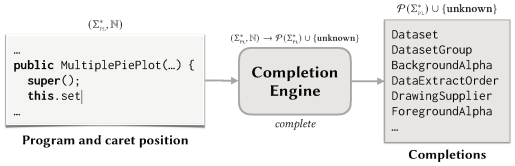


Figure 2: Abstraction of a Completion Engine.

Definition 3.4 (Strict Completion Engine). Assume that a Completion Engine **CE** can obtain a set of completions given a program *prog* feasible at *caret* (i.e., $(prog, caret) \vDash \Phi$):

$$\begin{gathered} completions = complete(prog, caret) \\ \text{where } completions \neq \textbf{unknown}. \end{gathered} \tag{3.6}$$

Then, **CE** is said to be strict if and only if, under this condition, continuing *prog* with any code that does not match with this set of completions yields an infeasible program at the new caret position:

$$\begin{gathered} \forall c \notin \text{Prefix}(completions), (prog', caret') \nvDash \Phi, \\ \text{where } prog' = prog\,[caret \leftarrow c] \text{ and } caret' = caret + |c|, \\ \text{Prefix}(\cdot) = \{c \mid s \in \cdot \text{ and } c \text{ is a prefix of } s \text{ or vice versa}\}. \end{gathered} \tag{3.7}$$

This definition essentially means that a strict Completion Engine should not give incorrect suggestions. It should return **unknown** whenever unsure. A trivial strict Completion Engine can be the one that always returns **unknown**.

3.3 Abstraction of LLMs

In this section, we give a formal abstraction of an encoder-decoder based LLM as showed in Figure 3, which in practice is more complex but conforms to the abstraction. The abstraction subsumes decoder-only models and can also describe encoder-only models that use the encoder outputs directly as token probabilities for generation.

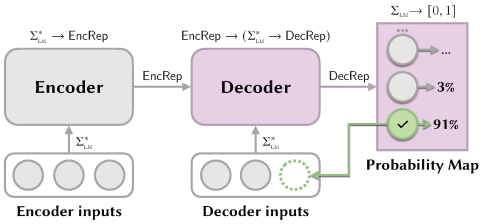


Figure 3: Abstraction of encoder-decoder based LLM.

Definition 3.5 (Large Language Model). Formally, We define an encoder-decoder based LLM **LM** as a 3-tuple

$$\mathbf{LM} = (\Sigma_{\text{LM}}, \mathit{encode}, \mathit{decode}), \tag{3.8}$$

where Σ_{LM} is a *vocabulary* consisting of the set of tokens defined by the model. The encoder *encode* is a function that maps from an input sequence to its encoded representation in EncRep:

$$\mathit{encode}\colon \Sigma_{\text{LM}}^{*} \longrightarrow \text{EncRep}. \tag{3.9}$$

The decoder *decode*, defined below, then *memorizes* the encoded representation in EncRep, takes as input a sequence of tokens, and produces as output its decoded representation in DecRep:

$$\mathit{decode}\colon \text{EncRep} \longrightarrow \left(\Sigma_{\text{LM}}^{*} \longrightarrow \text{DecRep}\right). \tag{3.10}$$

In this definition, the decoder memorizing the encoded representation is modeled as a *higher-order function* that returns a detailed decoding function given the encoded representation. The decoded representation in DecRep essentially assigns a probability to each token in the vocabulary to state its likelihood of being the next token in the sequence. Therefore, we can *define* DecRep as

$$\text{DecRep} = \Sigma_{\text{LM}} \longrightarrow [0, 1]. \tag{3.11}$$

4 APPROACH

Following most recent deep learning based APR tools [67, 72, 74], Repilot focuses on fixing single-hunk bugs, where the patch is obtained by changing a continuous section of code under perfect fault localization. Repilot can be extended for multi-hunk bugs by replacing all hunk locations at the same time with separate infilling tokens and using LLM to generate the replacement hunks. Benefiting from the era of LLMs, as shown in Figure 4, in this paper, we treat the repair problem as a *cloze* task [67], where a patch is formed by first replacing the buggy hunk with a masked span token (<SPAN>) and then using the LLM to direct synthesize the fixed hunk from the surrounding code context to replace the span token.

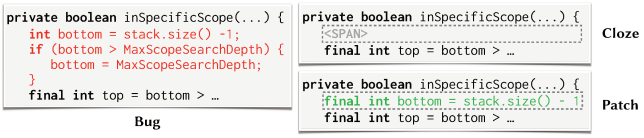


Figure 4: Cloze-style program repair.

4.1 Overview

Figure 5 shows an overview of how Repilot synthesizes a program that acts as the repaired hunk of the original buggy program. The generation loop consists of a loop that keeps updating the generation with tokens newly generated from the synergy between the language model and Completion Engine. The loop starts by applying the current generation as the input to the language model (❶), which returns a search space of a mapping from a suggested next token to its probability. Repilot then enters a token selection phase that repeatedly samples a token from the search space, checking its feasibility, and pruning the search space until a token is accepted. Every time a token is sampled, Repilot first checks if it hits the memorization (❷), which stores the tokens that are known to be feasible or infeasible. The memorization of infeasible tokens includes the use of a prefix tree data structure (Trie) discussed in §4.3. When the token hits the memorization and is infeasible, the search space is pruned by setting this token's probability to zero (❸), and the next sampling will run on the updated search space. In this way, the same token is not sampled again during the token selection phase. If the token misses the memorization, the search space is pruned under the guidance of the Completion Engine (❹), which we elaborate in §4.2. Provided that the sampled token is rejected by the Completion Engine, Repilot zeroes out its probability. Otherwise, it is accepted and this token selection process terminates. The memorization gets updated in both cases (❺). After a token is accepted (❻), we further leverage the Completion Engine, trying to actively complete the token (❼). The active completion, discussed in §4.4, may either produce more tokens or add nothing to the accepted token. Finally, Repilot appends all the newly generated tokens to the current generation and begins a new loop until a complete patch is generated. The loop stops when the model generates the special token **end-token**.

Algorithm 1 details this process and shows how a complete patch program is generated using what is established in §3. It additionally describes how Repilot performs the pre-processing (Lines 3 to 6) and formalizes completion-guided pruning procedure illustrated in Figure 5 using two functions **GuidedPrune** and **Actively-Complete** (Lines 7 to 17). In all our algorithms, we use a "dot-notation" to specify an entity of a tuple (e.g., **LM**.*encode*), but use an abbreviation form when the context is clear (e.g., Σ_{LM} and Σ_{PL} for **LM**.Σ_{LM} and **CE**.Σ_{PL}). We also optionally apply *type annotations* for clarification. Note that we simplify the definition of the Completion Engine by restricting it to be called with one program. In practice, a Completion Engine is always initialized with the entire project and can provide suggestions based on global information.

4.2 Completion-Guided Search Space Pruning

In this section, we explain the core idea of how Repilot utilizes a Completion Engine to prune the search space of an LLM.

Algorithm 2 explains in depth how a Completion Engine helps prune the model's search space. The function **GuidedPrune** takes as inputs a Completion Engine **CE**, the current program *prog*, the current caret position *caret*, and the probability map *tokens* given by the model, and produces a token *next-token* as the continuation of the program *prog* at position *caret*. The function consists of a **while**-loop (Lines 2 to 11) where Repilot first samples a possible next token according to the probabilities (Line 3), updates the current program accordingly (Line 4), and moves the caret after *next-token*. Repilot then invokes the Completion Engine using the function *complete* defined in Equation (3.5), given the program *prog*′ and the caret position *caret*′. If the result is not **unknown** but there is no completion (Line 8), it means that no possible continuation can

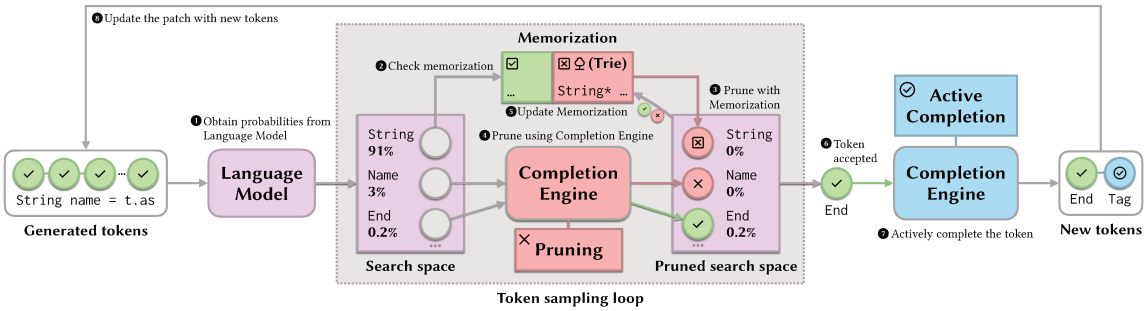


Figure 5: Overview of Repilot.

Algorithm 1 Main Repair Loop of Repilot

Inputs: Large Language Model **LM**, Completion Engine **CE**, Buggy program *prog*, and Range of buggy hunk *range*.
Output: Patch for the buggy program.

```
1:  func REPAIR(LM, CE, prog: Σ*_PL, range: ℕ × ℕ) → Σ*_PL:
2:      ▷ Initializations based on Definition 3.5 ◁
3:      encoder-inputs: Σ*_LM := BUILDINPUTS(prog, range)
4:      encoded-rep: EncRep := LM.encode(encoder-inputs)
5:      decoder: Σ*_LM → DecRep := LM.decode(encoded-rep)
6:      hunk: Σ*_LM := ε
7:      while true do
8:          ▷ Form patch by replacing buggy hunk with hunk ◁
9:          patch := prog[range ← STR(hunk)]
10:         ▷ Move caret after the current generation ◁
11:         caret := range.start + |STR(hunk)|
12:         tokens: Σ_LM → [0, 1] := decoder(hunk)
13:         next-token: Σ_LM := GUIDEDPRUNE(CE, patch, caret, tokens)
14:         if next-token = end-token then
15:             return patch
16:         completion-toks: Σ*_LM := ACTIVELYCOMPLETE(CE, patch, caret)
17:         hunk := hunk · next-token · completion-toks
```

be formed after *next-token*, so the token *next-token* is considered infeasible, thus *pruned* (Line 9) in this round of search, and the loop will continue (Line 10). Otherwise, we consider the token feasible and return *next-token* (Line 11).

The pruning at Line 9 is done by setting the probability of the entry *next-token* of the probability map *tokens* to zero. The notation used at this line is defined subsequently. Assume that

$$f : \mathsf{X} \to \mathsf{Y} = \{x_0 \mapsto y_0, x_1 \mapsto y_1, \dots\} \tag{4.1}$$

is an arbitrary function, and

$$a : \mathsf{X} \rightharpoonup \mathsf{Y} = \{x'_0 \mapsto y'_0, x'_1 \mapsto y'_1, \dots\} \tag{4.2}$$

is a *partial function* of the same type, meaning that only a subset of inputs in the domain X is associated with an output in the range Y. We define the action of changing the output values of the inputs in f using the assignments given by a as

$$\begin{gathered} f\,[a] = (f - f_{\text{removed}}) \cup a \\ \text{where } f_{\text{removed}} = \{x' \mapsto f(x') \mid x' \mapsto y' \in a\}. \end{gathered} \tag{4.3}$$

4.3 Memorization for Faster Search

Algorithm 2 (**GUIDEDPRUNE**) involves a loop of trials and pruning actions, which slows down the repair task in some situation. To

Algorithm 2 Completion-Guided Search Space Pruning

Inputs: Completion Engine **CE**, Current Program *prog*, Caret Position *caret*, and Token Probability Map *tokens*.
Output: Next token *next-token* to generate.

```
1:  func GUIDEDPRUNE(CE, prog, caret, tokens: Σ_LM → [0, 1]) → Σ_LM:
2:      while true do
3:          next-token: Σ_LM := SAMPLE(tokens)
4:          prog' := prog[caret ← STR(next-token)]
5:          caret' := caret + |STR(next-token)|
6:          ▷ completions: P(Σ*_PL) ∪ {unknown} ◁
7:          completions := CE.complete(prog', caret')
8:          if completions ≠ unknown and |completions| = 0 then
9:              tokens := tokens[{next-token ↦ 0}]
10:             continue
11:         return next-token
```

speedup its search procedure, we apply several memorization techniques to reduce the frequency of invoking the Completion Engine for analysis.

Memorizing rejected tokens. To repair a bug in practice requires generating plenty of samples, meaning that the same program *prog'* and *caret'* (Lines 4 to 5) may occur repeatedly in Algorithm 2 (**GUIDEDPRUNE**). Therefore, we can memorize all the tokens pruned at Line 9 by storing them in a variable

$$\textit{rejected} : (\Sigma^*_{\text{PL}}, \mathbb{N}) \to \mathcal{P}(\Sigma_{\text{LM}}), \tag{4.4}$$

which maps from a program *prog* and a caret position *caret* to a set of rejected tokens. Then we zero the probabilities of the rejected tokens in advance, written as

$$\textit{tokens} := \textit{tokens}\,[\{\textit{tok} \mapsto 0 \mid \textit{tok} \in \textit{rejected}(\textit{prog}, \textit{caret})\}], \tag{4.5}$$

before the **while**-loop (Line 2) starts.

Memorizing accepted tokens. Besides rejected tokens, we can also memorize tokens that are accepted before in a variable

$$\textit{accepted} : (\Sigma^*_{\text{PL}}, \mathbb{N}) \to \mathcal{P}(\Sigma_{\text{LM}}) \tag{4.6}$$

to avoid the overhead incurred from querying the Completion Engine at Lines 7 to 8.

Building a Prefix Tree of Rejected Tokens. It is common that many tokens in the vocabulary of the language model are prefixes of another. And it is obvious that if a token is rejected, meaning that no possible continuation can be formed after the token to obtain a statically valid program, then any token sharing such prefix should be rejected. For this reason, we build and keep updating a prefix tree, or Trie, of all the rejected tokens given *prog* and *caret*, and

checks if any of the tokens in the Trie is a prefix of *next-token* right after Line 3 in Algorithm 2. If it is the case, Repilot directly skips to the next iteration, avoiding further analysis.

4.4 Active Completion

Not only is a Completion Engine able to determine the feasibility of a possible next token suggested by the model, as shown in §4.2, but it can also *proactively* suggest a potential continuation of the current program without querying the model, just like how developers benefit from autocompletion.

Algorithm 3 describes active completion in detail. The function **ActivelyComplete** takes three inputs: the Completion Engine **CE**, the current program *prog*, and the current caret position *caret*, and outputs a sequence of tokens *completion-toks* as the continuation of *prog* at *caret*. Initially, Repilot gets the completion result according to Equation (3.5), given *prog* and *caret* (Line 2), and checks if it is **unknown** (Line 3). If it is the case (*completions* = **unknown**), the result is set to an empty string, meaning no extra completions are produced (Line 4). Otherwise, Repilot calculates the *common prefix* of all the completions (Line 6). Note that the type of the resultant variable *completion* is a sequence of characters in the Programming Language alphabet, different from the language model's Σ_{LM}, so Repilot further aligns the completion to fit the model's vocabulary (Line 7). Finally, the result is returned at Line 8.

Algorithm 3 Active Completion

Inputs: Completion Engine **CE**, Program *prog*, and Caret Position *caret*.
Output: The actively completed tokens *completion-toks*.

1: **func** ActivelyComplete(CE, *prog*, *caret*) $\rightarrow \Sigma^*_{\text{LM}}$:
2: $\quad$ *completions*: $\mathcal{P}(\Sigma^*_{\text{PL}}) \cup \{\textbf{unknown}\}$:= CE.*complete*(*prog*, *caret*)
3: $\quad$ **if** *completions* = **unknown** **then**
4: $\quad\quad$ *completion-toks* := ε
5: $\quad$ **else**
6: $\quad\quad$ *completion*: Σ^*_{PL} := CommonPrefix(*completions*)
7: $\quad\quad$ *completion-toks*: Σ^*_{LM} := AlignTokens(Σ_{LM}, *completion*)
8: $\quad$ **return** *completion-toks*

4.5 Soundness of Repilot

In this section, we show the theoretical guarantee of each algorithm discussed above under the condition that the Completion Engine is *strict* (Definition 3.4).

Lemma 4.1 (Soundness of Pruning). *The tokens pruned away in Algorithm 2 (**GuidedPrune**) result in infeasbile programs.*

Discussion. From Equation (3.7) in Definition 3.4, we can deduce that a program is infeasible at some caret position if the Completion Engine does not return **unknown** but the set of completions is empty, i.e.,

$$|\textit{completions}| = 0 \rightarrow (\textit{prog}, \textit{caret}) \notin \Phi \quad \text{if } \textit{completions} \neq \textbf{unknown} \tag{4.7}$$

The pruning at Algorithm 2 happens at Lines 8 to 9, which is exactly what is described above. As a result, we can conclude that the program with *next-token* appended is infeasible, and hence it is safe for Repilot to abandon the token. □

Lemma 4.2 (Soundness of Memorization). *The memorization discussed in §4.3 does not affect **GuidedPrune**'s behavior.*

Discussion. The theorem holds because all the memorization techniques mentioned in §4.3 do not change the semantics of **GuidedPrune** but only speed up the process. □

Lemma 4.3 (Soundness of Active Completion). *If a program is feasible at some caret position, the new program produced by Algorithm 3 (**ActivelyComplete**) is feasible at its new caret position.*

Discussion. Based on Equation (3.7) from Definition 3.4, any continuations not matching the set of completions would bring about an infeasible program. In the case where these completions have a shared common prefix, any continuations not starting with this common prefix would be invalid. Therefore, completing the original program with the common prefix (Line 6 in Algorithm 3) is the only way to yield a new feasible program. □

On the basis of Lemmas 4.1 to 4.3, we can easily prove that Repilot's overall algorithm is sound.

Theorem 4.4 (Overall Soundness). *Algorithm 1 (**Repair**) does not miss any feasible programs in the language model's search space.*

When will Repilot fail? Although the theorems are about the soundness of Repilot, i.e., it *prunes the search space correctly*, it does not provides any guarantee that Repilot *produces a valid patch* every time. Therefore, Repilot's expected behavior is to be able to obtain valid patches more efficiently, rather than being entirely error-free during the generation.

5 EXPERIMENTAL SETUP

In this paper, we study the following research questions to evaluate Repilot.

- **RQ1:** How does Repilot's bug fixing capability compare with state-of-the-art APR techniques (§6.1)?
- **RQ2:** How effective is Repilot in improving the compilation rate of patch generation (§6.2)?
- **RQ3:** Are all components of Repilot making positive contributions to its effectiveness (§6.3)?
- **RQ4:** Can Repilot generalize to different subjects of bugs and models (§6.4)?

We first compare the repair performance of Repilot, instantiated with CodeT5, against state-of-the-art APR tools across both traditional, NMT-based, and LLM-based tools on the Defects4J datasets in RQ1. In RQ2, we then closely evaluate the improvement in compilation rate — percentage of compilable patches generated to demonstrate that Repilot is not only effective in bug repair but can generate a higher number of compilable patches compared with existing tools. Furthermore, we perform a detailed ablation study in RQ3 to evaluate the contribution of different components of Repilot. Finally, in RQ4, we extend our evaluation of Repilot beyond its use with CodeT5 in the previous RQs. We go a step further by implementing Repilot with InCoder and assessing the performance of Repilot using both CodeT5 and InCoder on single-hunk bugs from both Defects4J 1.2 and 2.0 to demonstrate the generalizability of Repilot across different LLMs and bug subjects.

5.1 Implementation

We use the Python implementation of the CodeT5-large and the InCoder-6.7B models obtained on Hugging Face [26]. We build our generation pipeline in Python with 5K lines of code and implement a modified version of the Eclipse JDT Language Server [1, 18] in Java with 1.5K additional lines of code, which serves as the *strict* Completion Engine of our framework. Our default generation uses top-p nucleus sampling [24] with $p = 1.0$, *temperature* = 1.0, *max-tokens* = 50 and samples 5000 times per bug for fair comparisons against prior works (§6.1 and §6.2). Due to the high cost of APR, we sample 500 times per bug for the ablation study (§6.3) and the generalizability evaluation (§6.4). Following prior work [39, 44, 67, 74], we use a timeout of 5 hours to generate and validate all patches per bug. We generate and validate patches on a 32-Core with Ryzen Threadripper PRO 3975WX CPU, 256 GB RAM and NVIDIA RTX A6000 GPU, running Ubuntu 20.04.4 LTS with Java version OpenJDK 1.8.0_181.

5.2 Subject Programs

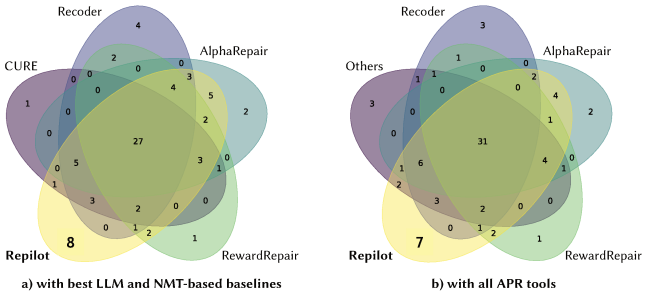
We use the popular repair benchmark of Defects4J for our evaluation. Defects4J is a manually curated Java dataset with pairs of buggy and patched versions of the source project along with developer test suites for validation. Following prior work and APR literature convention, we separate Defects4J into Defects4J 1.2, containing 391 bugs (removing 4 depreciated bugs) from 6 Java source projects, and Defects4J 2.0, containing 438 new bugs from 9 additional projects. For Defects4J 1.2, we focus on only the single-hunk bugs as Repilot is designed for single-hunk repair. Note this is also the evaluation setting used in the prior baseline [72]. Furthermore, we remove the bugs that are incompatible with our Completion Engine due to engineering issues. In total, we consider 138 single-hunk bugs from Defects4J 1.2 and 135 single-hunk bugs from Defects4J 2.0. For our main evaluation in RQ1, following the same setup as prior LLM for APR work [66, 67], we report the results on all 135 single-hunk bugs from Defects4J 1.2 and 76 single-line bugs (a subset of single-hunk bugs) from Defects4J 2.0. Meanwhile, in our generalizability study (RQ4), we further evaluate Repilot on the full set of single-hunk bugs from both Defects4J 1.2 and 2.0 for both CodeT5 and InCoder.

5.3 Compared Techniques

We compare Repilot against state-of-the-art baselines from traditional, NMT-based, and LLM for APR tools. We evaluate against AlphaRepair [67] as it is the top performing LLM for APR approach. For NMT-based approaches, we choose 6 recent tools: RewardRepair [72], Recoder [74], CURE [29], CoCoNuT [44], DLFix [39] and SequenceR [11] based on the NMT architecture. Additionally, we compare against 12 traditional APR tools: PraPR [21], TBar [41], AVATAR [42], SimFix [28], FixMiner [34], CapGen [63], JAID [9], SketchFix [25], NOPOL [13], jGenProg [45], jMutRepair [46] and jKali [46]. Altogether, we include 19 APR baselines and compare Repilot against them on Defects4J 1.2 and 2.0. Our evaluation setting is on perfect fault localization – where the ground-truth location of the bug is given to the APR tool. We note that this is the preferred evaluation setting as it eliminates any differences caused by different fault localization methods [29, 44, 58, 74]. We follow the convention used in prior work [21, 29, 41, 74] and directly report the bug fix results obtained from previous studies [21, 67, 72].

Table 1: Number of correct fixes on Defects4J 1.2 single-hunk and Defects4J 2.0 single-line bugs

Tool	Methodology	#Correct Fixes		
		Defects4J 1.2	Defects4J 2.0	Total
CoCoNuT	NMT	30	-	-
DLFix	NMT	32	-	-
PraPR	Template	35	-	-
TBar	Template	41	7	48
CURE	NMT	43	-	-
RewardRepair	NMT	45	24	69
Recoder	NMT	51	10	61
AlphaRepair	LLM	52	34	86
Repilot	LLM	**66**	**50**	**116**



a) with best LLM and NMT-based baselines b) with all APR tools

Figure 6: Correct fix Venn diagrams on Defects4J 1.2

5.4 Evaluation Metrics

- **Plausible patches** are patches that pass all test cases but may violate the real user intent.
- **Correct patches** are patches that are semantically equivalent to the developer patch. Following common APR practice, we determine semantic equivalency by manually examining each plausible patch.
- **Patch compilation rate** is also used in many deep learning based APR works [29, 72], which indicates the percentage of compilable patches in all generated patches.

6 RESULT ANALYSIS

6.1 RQ1: Comparison with Existing Tools

In RQ1 and RQ2, we follow the prior approach for cloze-style APR [67] to make use of repair templates for a faithful evaluation. Instead of replacing the entire buggy line with model-generated code, these templates systematically keep parts of the buggy line to reduce the amount of code the LLM needs to generate. Note that we do not apply any repair templates in RQ3 and RQ4 because we consider a smaller number of samples there (i.e., 500 samples as shown in Section 5.1) and also want to focus on the impact of different experimental configurations.

Defects4J 1.2. We first compare Repilot against the state-of-the-art APR tools on single-hunk bugs from Defects4J 1.2. Table 1 shows the number of correct patches produced by Repilot, evaluated in cloze-style, along with the baselines. *Repilot achieves the new state-of-the-art result of 66 correct bug fixes on Defects4J 1.2, outperforming*

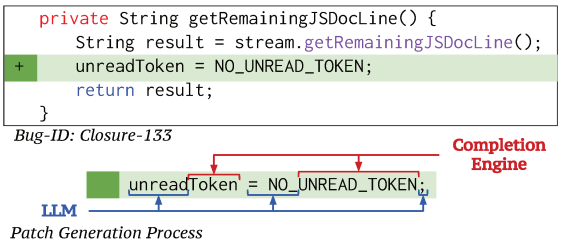


Figure 7: Unique bug fix by Repilot on Defects4J 1.2

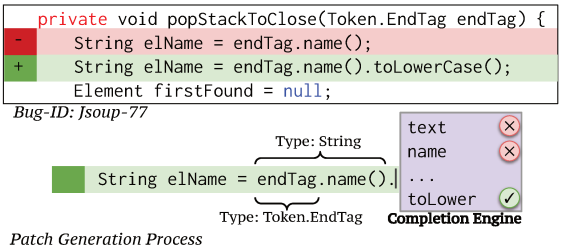


Figure 8: Unique bug fix by Repilot on Defects4J 2.0

all previous APR tools. Figure 6a shows the Venn diagram of the unique bugs fixed for the top performing LLM- and NMT-based APR tools where Repilot is able to obtain the highest number of 8 unique bugs Furthermore, Figure 6b compares the unique bugs fixed for all top-performing baselines and with all other APR tools combined (Others). We observe that Repilot is able to fix 7 bugs that no other baselines have been able to fix so far.

To demonstrate the ability of Repilot to fix difficult bugs, Figure 7 shows a unique bug (`Closure-133`) from Defects4J 1.2 that only Repilot can fix. This bug is fixed by adding the new assignment statement using the global variable `NO_UNREAD_TOKEN` which is difficult to generate as it does not appear within the surrounding context of the bug location. Repilot first uses CodeT5 to generate the initial prefix of `unread`. Then using the Completion Engine, Repilot recognizes that `Token` is the only semantically correct continuation and directly performs active completion to return `unreadToken`. Similarly for generating `NO_UNREAD_TOKEN`, Repilot first generates `NO_` and then uses active completion to directly generate this rare identifier without having to repeatedly sample the LLM. It is difficult for prior LLM- and NMT-based APR tools to generate this fix as LLMs or NMT models may not be able to complete this rare identifier since it requires multiple continuous steps to generate. In contrast, Repilot, through the use of active completion, can directly generate this rare identifier given only the initial identifier prefix to quickly arrive at this correct patch.

Defects4J 2.0. We further evaluate Repilot against baselines evaluated on the single-line bugs in Defects4J 2.0. For these bugs, we follow prior approach for cloze-style APR [67] to make use of repair templates. Instead of replacing the entire buggy line with model-generated code, these templates systematically keep parts of the buggy line (e.g., prefix or suffix, method parameters and calls) to reduce the amount of code the LLM needs to generate. We apply these repair templates for Defects4J 2.0 single-line bugs only since they are designed for single-line bugs. Table 1 also shows the number of correct fixes on Defects4J 2.0 compared with the baselines. We observe that *Repilot is able to fix the highest number of bugs 50 (16 more than the next best baseline) on Defects4J 2.0.* This improvement over existing baselines shows that Repilot can generalize to two versions of Defects4J datasets and demonstrates the power of repair templates to boost the performance of LLM-based APR tools.

Figure 8 shows a unique bug from Defects4J 2.0 that only Repilot can fix. First, Repilot generates the patch up to the caret position. The Completion Engine then captures the exact type of the object from `Token.EndTag` to `String`. Using this information, Repilot correctly prunes tokens that are not a part of the `String` class (e.g., `name` and `text`). Hence, the generated patch contains a valid `String` class method of `toLowerCase()` which correctly fixes this bug. Similar to the previous unique bug fix in Defects4J 1.2, prior LLM-based APR tools may waste a lot of time generating semantically incorrect continuations as they do not have access to the type information. Furthermore, NMT-based APR tools such as CURE [29], over-approximating the list of valid identifiers by statically grabbing all the accessible fields, may not generate this fix since a pruned identifier (e.g., `name`) can also be valid for a different object type. Repilot uses the Completion Engine to analyze partial programs and realize complex type propagation for effective pruning.

Table 2: Comparison with existing APR tools on compilation rate on Defects4J 1.2. "-" denotes data not available.

Tool	% Compilable Patches			
	Top-30	Top-100	Top-1000	Top-5000
SequenceR	33%	-	-	-
CoCoNuT	24%	15%	6%	3%
CURE	39%	28%	14%	9%
AlphaRepair	25%	22%	16%	13%
RewardRepair	45%	38%	33%[1]	-
Repilot	**66%**	**62%**	**58%**	**59%**

[1] This is the top 200 rate for RewardRepair as it does not include top 1000

6.2 RQ2: Compilation Rate Analysis

We evaluate the compilation rate of the patches generated by Repilot compared with prior learning-based APR techniques. Table 2 shows the percentage of compilable patches on the Defects4J 1.2 dataset. We observe that across all numbers of patches generated, Repilot significantly improves the percentage of compilable patches compared with prior tools. We first notice that LLM-based APR tools (Repilot and AlphaRepair), are able to sustain their compilation rate compared with NMT-based tools (CoCoNuT and CURE) where the compilation rate drastically decreases as we increase the number of patches. This shows the ability for LLMs to generate large amounts of reasonable patches. Repilot is able to sustain a near 60% compilation percentage at 1000 patches generated while the prior approach is barely above 30%.

Compared with CURE [29], where an overestimation of valid identifiers is obtained via static analysis and used to prune invalid tokens generated by NMT model, Repilot leverages the powerful Completion Engine to keep track of the current context to obtain a more accurate pruning step. Furthermore, compared with RewardRepair [72], where the compilation rate is boosted through penalizing uncompilable patches during training, Repilot directly uses a LLM combined with a Completion Engine to avoid this high

Table 3: Component contribution of Repilot

Variant	Generation Time	%Compilable Patches	%Plausible Patches	#Plausible Fixes	#Correct Fixes
$\text{Repilot}_{\varnothing}$	0.232s	43.2%	3.95%	56	37
$\text{Repilot}_{\text{P}}$	0.294s	60.7%	5.02%	62	41
$\text{Repilot}_{\text{P}}^{\text{M}}$	0.255s	58.7%	4.82%	60	40
Repilot	**0.248s**	**63.4%**	**5.21%**	**63**	**42**

cost of training a new model. Additionally, Repilot uses the active completion ability of Completion Engine to directly generate these rare identifiers to further boost the compilation rate. As such, Repilot is able to achieve the highest percentage of compilable patches across all four different settings.

6.3 RQ3: Ablation Study

To study the contribution of each component of Repilot to its overall effectiveness, we conduct an ablation study that aims at justifying the following hypothesis:

- Algorithm 2 (**GuidedPrune**) helps LLM to achieve valid (compilable) patches more efficiently on a pruned search space.
- Memorization (§4.3) reduces the frequency of querying the Completion Engine, thus speeding up patch synthesis.
- Active completion provides further guidance of synthesis that and helps Repilot efficiently achieve more valid patches.
- The plausible rate of patches becomes higher along with the compilation rate.

To give grounds for these hypotheses, we set up the following four variants:

- $\text{Repilot}_{\varnothing}$ uses only the base LLM (CodeT5) for patch synthesis.
- $\text{Repilot}_{\text{P}}$ applies pruning defined in Algorithm 2.
- $\text{Repilot}_{\text{P}}^{\text{M}}$ leverages memorization (§4.3) on top of pruning.
- Repilot employs active completion for further guidance.

and evaluate them by comparing them against their efficiency in generating compilable, plausible patches, and correct patches.

Table 3 shows the generation time (in seconds per patch), the contribution in terms of the percentage of compilable and plausible patches among all uniquely generated patches, the number of plausible fixes, and the number of correct fixes for each of the four variants on Defects4J 1.2 single-hunk bugs. We first observe that just using the base LLM for APR ($\text{Repilot}_{\varnothing}$), we achieve the lowest compilation rate at 43.2%. By adding the pruning provided by the Completion Engine, we can significantly improve the compilation rate to 60.7%, the number of plausible fixes from 56 to 62, and the number of correct fixes from 37 to 41. Additional improvement is made by adding the active completion technique to achieve the full Repilot with the highest compilation rate at 63.4%, plausible percentage 5.21%, the most number of plausible fixes at 63, and the most correct fixes at 42.

Looking at the patch generation time, starting from $\text{Repilot}_{\varnothing}$, adding pruning via Completion Engine incurs an over 25% overhead. However, this can be significantly reduced by using memorization ($\text{Repilot}_{\text{P}}$) to achieve around 10% overhead by avoiding querying the Completion Engine once we know an identifier is invalid. Furthermore, active completion can further reduce the overhead to 7% since instead of having to sample the LLM for each step in the generation, we can actively complete an identifier.

As a result, all the components contribute to the overall effectiveness of Repilot. Repilot can consistently increase the compilation and plausible rate, as well as produce more plausible/correct fixes while incurring minimal overhead compared with directly using LLMs for patch synthesis.

6.4 RQ4: Generalizability

To demonstrate the generalizability of Repilot across different subjects of bugs and models, on the one hand, we further evaluate Repilot with CodeT5 on all single-hunk bugs of Defects4J 2.0. On the other hand, we additionally instantiate and evaluate Repilot with a larger InCoder-6.7B model. Identical to RQ3, we also conduct 500 samples in RQ4 due to the high cost of APR.

Table 4 shows the comparison between the baseline $\text{Repilot}_{\varnothing}$ and our full Repilot approach across different subjects of bugs and models. We consider the same set of Defects4J 1.2 single-hunk bugs as in RQ3 and an extra set of Defects4J 2.0 single-hunk bugs.

Upon investigation, we can see that Repilot with CodeT5 surpasses the baseline on Defects4J 1.2 as illustrated in RQ3. Furthermore, on Defects4J 2.0, it can also achieve 18.1 percentage points (pp) more compilable and 3.0 pp more plausible patches, as well as 6 more plausible fixes and 4 more correct fixes, with a 7.4% overhead.

Meanwhile, when Repilot is instantiated with InCoder, it still produces more compilable and plausible patches, as well as more plausible and correct fixes on both Defects4J 1.2 and Defects4J 2.0 over the baseline InCoder. It eventually gives 6 more correct fixes on Defects4J 1.2 and 1 more on Defects4J 2.0.

One major difference comparing Repilot with InCoder and CodeT5 is that when Repilot is equipped with InCoder, a much larger model than CodeT5, it incurs negligible overhead. This is because compared to the high cost of autoregressive sampling using larger models, the extra cost from querying the Completion Engine is much smaller and thus trivializes the overhead of Repilot when applied on larger models. Also, the larger InCoder model, whether or not it is applied with Repilot, can consistently fix more bugs across both Defects4J 1.2 and 2.0 than CodeT5, further confirming prior finding that larger LLMs often perform better for APR [66].

Overall, the experimental results indicate that Repilot can generalize to different sets of bugs (both single-hunk bugs in Defects4J 1.2 and 2.0) as well as larger LLMs (InCoder)

7 LIMITATIONS

First, to bring out Repilot's full potential, it is important that the Completion Engine can provide useful guidance while remaining strict (Definition 3.4). However, it is generally more difficult to balance the usefulness and strictness of a Completion Engine in many dynamically typed programming languages, such as Python, compared with Java studied in this paper, which is a statically typed programming language. Meanwhile, there is a growing trend of dynamically typed languages adopting support for type hints [12, 49, 54]. Considering this, we believe that Repilot can still provide significant advantages in such environments.

Another limitation of Repilot lies in the evaluation. On the one hand, while it is true that an increase in the compilation rate of

Table 4: Generalizability of Repilot across both subjects of bugs and models

Variant	Model	Subject of Bugs	Generation Time	%Compilable Patches	%Plausible Patches	#Plausible Fixes	#Correct Fixes
Repilot$_{\varnothing}$	CodeT5-large	Defects4J 1.2	**0.232s**	43.2%	3.95%	56	37
Repilot	CodeT5-large	Defects4J 1.2	0.248s	**63.4%**	**5.21%**	**63**	**42**
Repilot$_{\varnothing}$	CodeT5-large	Defects4J 2.0	**0.230s**	46.7%	9.02%	59	41
Repilot	CodeT5-large	Defects4J 2.0	0.247s	**64.8%**	**12.02%**	**65**	**45**
Repilot$_{\varnothing}$	InCoder-6.7B	Defects4J 1.2	**1.70s**	32.4%	3.85%	70	48
Repilot	InCoder-6.7B	Defects4J 1.2	**1.70s**	**47.2%**	**4.96%**	**78**	**54**
Repilot$_{\varnothing}$	InCoder-6.7B	Defects4J 2.0	**1.67s**	34.6%	5.06%	67	45
Repilot	InCoder-6.7B	Defects4J 2.0	1.69s	**48.0%**	**6.87%**	**68**	**46**

Repilot can lead to the discovery of more plausible and correct fixes, it is important to note that a significantly higher compilation rate does not necessarily translate to a proportionally large increase in plausible and correct fixes. On the other hand, Repilot is only evaluated with CodeT5 for RQ1 and RQ2 with a 5000 sampling budget. CodeT5 is a rather "small" LLM compared to those LLMs with billions of parameters. Although we further include InCoder-6.7B as a multi-billion-parameter LLM in RQ4, due to time cost, we only sample 500 times per bug, which may be insufficient to reflect the distribution of the generated patches. Overall, the scope of our evaluation considering two LLMs (CodeT5 and InCoder) and one programming language (Java) is still narrow given that Repilot is a general framework that can be instantiated with any pair of an LLM and a Completion Engine for some programming language.

Finally, despite the examples we show in the paper, our evaluation lacks strong empirical evidence to support the claim that LLMs have difficulty in generating rare tokens and how Repilot solves the problem. Besides, our evaluation limits the application of Repilot to patch synthesis, even though we claim that Repilot can be applied to other code generation tasks. In the future, we will apply and evaluate Repilot on more diverse code generation tasks.

8 THREATS TO VALIDITY

Internal. We share the same main internal threat to validity with prior APR tools where we have to manually examine each plausible patch to determine patch correctness. We address this by carefully analyzing each patch to determine if it is semantically equivalent to the reference developer patch. Furthermore, we have released our full set of correct patches for public evaluation [62].

Our use of the CodeT5 model poses another internal threat where the open-source training dataset of GitHub projects [27] may overlap with our evaluation of Defects4J. We follow prior work [66, 67] and address this by computing the correct bug fixes of Repilot from Defects4J that is part of the CodeT5 training data. In total, 7 out of 66 and 6 out of 50 overlap with training data on Defects4J 1.2 and 2.0 respectively. For comparison fairness, if we were to exclude these 7 and 6 bugs and compare them with the previous baseline tools on the remaining bugs, we are still able to achieve the highest bug fixes at 59 and 44 (best baseline at 45 and 29). The same threat applies to the use of InCoder, but since its detailed training data is not revealed, we are unable to explicitly address this problem. To mitigate the problem, we only evaluate InCoder in RQ4, where all the variants face the same potential leakage.

Moreover, our modified implementation of the completion engine requires manual inspection to guarantee soundness property. In practice, this is a significant trust base that may introduce false positives during pruning. However, our theorem still provides a partial guarantee and is able to explain unsoundness. At the same time, our evaluation result justifies our claims and demonstrates the practicality of Repilot.

Finally, in our evaluation, we follow the convention used in prior work to directly report the bug fix results without reproducing them, which poses a threat to the reliability of the results. Meanwhile, we only run each of our experiments once, which could introduce extra statistical biases.

External. The main external threat to validity comes from our evaluation dataset where the performance of Repilot may not generalize to other datasets. To address this, we compare Repilot against state-of-the-art baselines on both Defects4J 1.2 and 2.0 to show that the performance is sustained across both versions. To address this further, we plan to evaluate Repilot on additional APR datasets also across different programming languages.

9 CONCLUSION

We propose Repilot — the first APR approach to combining the direct usage of LLMs (e.g., CodeT5 and InCoder) with on-the-fly guidance provided by Completion Engines. During autoregressive token generation, Repilot queries the Completion Engine not only to *prune* invalid tokens but also to *proactively complete* the currently generated partial program, thereby reducing the search space of the LLM. Our evaluation on a subset of the widely-studied Defects4J 1.2 and 2.0 datasets shows Repilot is able to achieve the state-of-the-art results. Furthermore, Repilot, through the usage of Completion Engine, is able to generate more valid and compilable patches than prior tools with minimal overhead compared with directly using LLMs for APR.

DATA AVAILABILITY

We have open-sourced Repilot, which can be accessed on GitHub at https://github.com/ise-uiuc/Repilot. Additionally, an immutable artifact for Repilot is publicly available on Zenodo [62].

ACKNOWLEDGMENTS

We thank all the reviewers for their insightful comments. We also thank Yifeng Ding for his helpful discussion on this work. This work was partially supported by NSF grants CCF-2131943 and CCF-2141474, as well as Kwai Inc.

REFERENCES

[1] 2023. Eclipse JDT LS. https://projects.eclipse.org/projects/eclipse.jdt.ls.

[2] Armen Aghajanyan, Bernie Huang, Candace Ross, Vladimir Karpukhin, Hu Xu, Naman Goyal, Dmytro Okhonko, Mandar Joshi, Gargi Ghosh, Mike Lewis, and Luke Zettlemoyer. 2022. CM3: A Causal Masked Multimodal Model of the Internet. *CoRR* abs/2201.07520 (2022). arXiv:2201.07520 https://arxiv.org/abs/2201.07520

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. arXiv:2103.06333 [cs.CL]

[4] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

[5] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 78 (apr 2023), 27 pages. https://doi.org/10.1145/3586030

[6] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The Plastic Surgery Hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 306–317. https://doi.org/10.1145/2635868.2635898

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[8] Jialun Cao, Meiziniu Li, Ming Wen, and Shing-Chi Cheung. 2023. A study on Prompt Design, Advantages and Limitations of ChatGPT for Deep Learning Program Repair. *CoRR* abs/2304.08191 (2023). https://doi.org/10.48550/ARXIV.2304.08191 arXiv:2304.08191

[9] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based program repair without the contracts. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 637–647. https://doi.org/10.1109/ASE.2017.8115674

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri 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, Josh 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. arXiv:2107.03374 [cs.LG]

[11] 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 Transactions on Software Engineering* 47, 9 (2021), 1943–1959. https://doi.org/10.1109/TSE.2019.2940179

[12] Clojure 2023. Typed Clojure: An Optional Type System for Clojure. https://typedclojure.org.

[13] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic Repair of Buggy If Conditions and Missing Preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis* (Hyderabad, India) *(CSTVA 2014)*. Association for Computing Machinery, New York, NY, USA, 30–39. https://doi.org/10.1145/2593735.2593740

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

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[16] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion. arXiv:2310.11248 [cs.LG]

[17] 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. *CoRR* abs/2002.08155. arXiv:2002.08155 https://arxiv.org/abs/2002.08155

[18] Eclipse Foundation and Yuxiang Wei. 2023. *UniverseFly/eclipse.jdt.ls: Modified Eclipse JDT LS 1.0.3*. https://doi.org/10.5281/zenodo.8278193

[19] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=hQwb-lbM6EL

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

[21] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair via Bytecode Mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 19–30. https://doi.org/10.1145/3293882.3330559

[22] GithubCopilot 2023. GitHub Copilot: Your AI pair programmer. https://github.com/features/copilot.

[23] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*. https://openreview.net/forum?id=jLoC4ez43PZ

[24] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rygGQyrFvH

[25] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. SketchFix: A Tool for Automated Program Repair Approach Using Lazy Candidate Generation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 888–891. https://doi.org/10.1145/3236024.3264600

[26] HuggingFace 2023. Hugging Face. https://huggingface.co.

[27] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019). arXiv:1909.09436 http://arxiv.org/abs/1909.09436

[28] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *ISSTA 2018* (Amsterdam, Netherlands). Association for Computing Machinery, New York, NY, USA, 298–309. https://doi.org/10.1145/3213846.3213871

[29] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[30] Yanjie Jiang, Hui Liu, Nan Niu, Lu Zhang, and Yamin Hu. 2021. Extracting Concise Bug-Fixing Patches from Human-Written Patches in Version Control Systems. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 686–698. https://doi.org/10.1109/ICSE43902.2021.00069

[31] Harshit Joshi, José Cambronero, Sumit Gulwani, Vu Le, Ivan Radicek, and Gust Verbruggen. 2023. *Repair Is Nearly Generation: Multilingual Program Repair with LLMs*. AAAI. https://www.microsoft.com/en-us/research/publication/repair-is-nearly-generation-multilingual-program-repair-with-llms/

[32] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[33] Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. 2022. Patch Generation with Language Models: Feasibility and Scaling Behavior. In *Deep Learning for Code Workshop*. https://openreview.net/forum?id=rHlzJh_b1-5

[34] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining Relevant Fix Patterns for Automated Program Repair. *Empirical Softw. Engg.* 25, 3 (may 2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z

[35] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 593–604. https://doi.org/10.1145/3106237.3106309

[36] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *SANER (2016)*, Vol. 1. 213–224. https://doi.org/10.1109/SANER.2016.76

[37] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[38] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. https://doi.org/10.1126/science.abq1158 arXiv:https://www.science.org/doi/pdf/10.1126/science.abq1158

[39] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 602–614. https://doi.org/10.1145/3377811.3380345

[40] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. arXiv:2305.01210 [cs.SE]

[41] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/3293882.3330577

[42] Kui Liu, Jingtang Zhang, Li Li, Anil Koyuncu, Dongsun Kim, Chunpeng Ge, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2023. Reliable Fix Patterns Inferred from Static Checkers for Automated Program Repair. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 96 (may 2023), 38 pages. https://doi.org/10.1145/3579637

[43] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 166–178. https://doi.org/10.1145/2786805.2786811

[44] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/3395363.3397369

[45] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4j Dataset. *Empirical Softw. Engg.* 22, 4 (aug 2017), 1936–1964. https://doi.org/10.1007/s10664-016-9470-4

[46] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 441–444. https://doi.org/10.1145/2931037.2948705

[47] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[48] Microsoft 2023. Language Server Protocol. https://microsoft.github.io/language-server-protocol.

[49] Microsoft 2023. TypeScript. https://www.typescriptlang.org.

[50] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. arXiv:2305.02309 [cs.LG]

[51] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. arXiv:2203.13474.

[52] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable Code Generation from Pre-trained Language Models. In *International Conference on Learning Representations*. https://openreview.net/forum?id=KmtVD97J43e

[53] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's Codex Fix Bugs? An Evaluation on QuixBugs. In *APR '22* (Pittsburgh, Pennsylvania). Association for Computing Machinery, New York, NY, USA, 69–75. https://doi.org/10.1145/3524459.3527351

[54] Python 2023. Type Hints in Python. https://peps.python.org/pep-0484/.

[55] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 1715–1725. https://doi.org/10.18653/v1/P16-1162

[56] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. arXiv:2301.08653 [cs.SE]

[57] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger (Eds.), Vol. 27. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf

[58] 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, Article 19 (sep 2019), 29 pages. https://doi.org/10.1145/3340544

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[60] 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:2305.07922 [cs.CL]

[61] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[62] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. ESEC/FSE'23 Artifact for "Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair". https://doi.org/10.5281/zenodo.8281250

[63] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3180155.3180233

[64] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. Revisiting the Plastic Surgery Hypothesis via Large Language Models. arXiv:2303.10494 [cs.SE]

[65] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal Fuzzing via Large Language Models. arXiv:2308.04748 [cs.SE]

[66] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[67] Chunqiu Steven Xia and Lingming Zhang. 2022. 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* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 959–971. https://doi.org/10.1145/3540250.3549101

[68] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational Automated Program Repair. *CoRR* abs/2301.13246 (2023). https://doi.org/10.48550/ARXIV.2301.13246 arXiv:2301.13246

[69] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. arXiv:2304.00385 [cs.SE]

[70] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3520312.3534862

[71] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. *XLNet: Generalized Autoregressive Pretraining for Language Understanding*. Curran Associates Inc., Red Hook, NY, USA.

[72] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-Based Backpropagation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1506–1518. https://doi.org/10.1145/3510003.3510222

[73] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. arXiv:2303.12570 [cs.CL]

[74] 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/FSE 2021* (Athens, Greece). Association for Computing Machinery, New York, NY, USA, 341–353. https://doi.org/10.1145/3468264.3468544

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