Memory-Efficient Large Language Models for Program Repair with Semantic-Guided Patch Generation

Anonymous Author(s)

ABSTRACT

Fixing software bugs is crucial yet demands significant resources from developers. Automated Program Repair (APR) is a promising solution to address this challenging task. The emergence of Large Language Models (LLMs) has opened a new era of LLM-based APR, substantially advancing the APR field further. LLM-based APR methods face significant challenges regarding memory inefficiency, hindering their scalability and effectiveness. This is largely due to the beam search utilized in the patch generation phase of LLM-based APR, which requires large beam sizes to search for more potentially good repair candidates.

In this paper, we first show that increases in beam size, even for small-sized LLMs (1B-7B params), require extensive GPU usage, leading to up to 80% of recurring crashes due to memory overloads in LLM-based APR. Seemingly simple solutions to reduce memory consumption are (1) to quantize LLM models, i.e., converting the weights of an LLM from high-precision values to lower-precision ones, and (2) to make beam search sequential, i.e., forwarding each beam through the model sequentially and then concatenating them back into a single output. However, we show that these approaches still do not work via both theoretical analysis and experiments.

To address this, we introduce FLAMES, a novel LLM-based APR technique that employs semantic-guided patch generation to enhance repair effectiveness and memory efficiency. Unlike conventional methods that rely on beam search, FLAMES utilizes greedy decoding to enhance memory efficiency while steering the search towards more potentially good repair candidates via a semantic-guided best-first search algorithm. At each decoding step, FLAMES uses semantic feedback from test validation, such as the number of passing and failing test cases, to select the most promising token to explore further. Our empirical evaluation on Defects4J shows that FLAMES substantially reduces memory consumption by up to 83% compared to LLM-based APR without compromising time efficiency. Moreover, FLAMES correctly fixes 133 bugs on Defects4J, fixing 10 bugs more than the best baseline. Additionally, these improvements also generalize to the HumanEval-Java and TransformedD4J datasets, where FLAMES generates 12% and 36.5% more correct patches, respectively, than the best baseline.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; **Software testing and debugging**.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, July 2017, Washington, DC, USA

© 2024 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

KEYWORDS

Program Repair, Memory Efficiency, Large Language Models

ACM Reference Format:
Anonymous Author(s). 2024. Memory-Efficient Large Language Models for Program Repair with Semantic-Guided Patch Generation. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Fixing software bugs is a complex and time-consuming task for developers [66]. Automated Program Repair (APR) [17, 19] has emerged as a promising solution to alleviate this burden by automatically repairing bugs. Over the past two decades, APR has seen significant advancements [9, 18, 35, 37, 40, 43, 72, 77, 86], with practical applications in the software industry [4, 32, 46, 67].

Recently, the surge of Large Language Models (LLMs), pre-trained on vast datasets, has further advanced APR, opening a new era of LLM-based APR [22, 57, 69]. LLM-based APR typically adapts models for APR either through fine-tuning on APR-specific datasets [20, 22, 53, 57, 81, 88] or by using prompting with commercial models [5, 80]. In this study, we focus on fine-tuning approaches, as prompting often relies on closed-source and commercial models like ChatGPT, which lack transparency, hindering reproducibility in open science and raising data privacy concerns.

Fine-tuning-based approaches typically refine the weights of Code LLMs by fine-tuning these models on APR datasets. During inference, they generate token sequences to construct candidate patches, prioritizing sequences with high probability scores from the fine-tuned model. Traditionally, beam search, a widely used search algorithm from natural language processing [59], is used to optimize this process by maintaining only the top-n nodes at each decoding step, where n is the beam size. A larger beam size provides a more precise approximation to exact decoding, i.e., thoroughly exploring all possible sequences, thereby enhancing the quality of the outputs. This hyperparameter is even more important in the context of APR, which covers a broader range of possible identifiers and a larger search space than natural language [9, 24, 45]. It has been shown that a substantial beam size is essential for generating adequate candidate patches for optimal results [24, 45, 85], and a larger beam size leads to higher performance of APR techniques that are based on small-sized deep learning models (less than 300M parameters) [62, 78, 85].

But, *is it true that increasing the beam size is all we need to improve the performance of APR?* We revisit the impact of beam size on the effectiveness of LLM-based APR techniques on larger models (1B-7B parameters). Our findings reveal that increasing the beam size boosts the effectiveness of these techniques, but once the beam size reaches a threshold, the performance drops significantly due to memory overloads that cause recurring crashes even on state-of-the-art hardware (NVIDIA A100 GPU, equipped with 80 GB of

VRAM). For instance, state-of-the-art LLM-based APR techniques generated between 21% and 46% more plausible patches when the beam size was increased from 10 to 25. Ideally, we expect that further increasing the beam size would lead to better performance of the models for APR. However, when the beam size is set higher at 50, 100, and 200, our experiments reveal that the performance of these models drops significantly.

Our experiments uncover that the above phenomenon is due to extensive GPU resource consumption, particularly on Video Random Access Memory (VRAM), caused by larger beam sizes. Often, upgrading VRAM to accommodate larger beam sizes requires GPU upgrades, which are expensive; thus, this creates a costly trade-off between memory efficiency and the performance of LLM-based APRs. For instance, achieving a 46% performance improvement with InCoder-1B required 58% more average and 111% more peak VRAM usage on our dataset. Consequently, further performance gains through increasing beam size demand substantial VRAM, often leading to out-of-memory (OOM) crashes even on cutting-edge hardware and thereby unduly reducing the effectiveness of LLM-based APR techniques. Our experiments using the NVIDIA A100 GPU, equipped with 80 GB of VRAM, showed that InCoder-6B crashed on nearly 80% of the evaluated bugs when using a beam size of 200. This resulted in a 60% reduction in performance compared to a beam size of 10. In summary, we conclude that simply increasing the beam size is not a solution, as it comes at the cost of chasing after state-of-the-art hardware.

A seemingly straightforward approach to this problem is to reduce the memory usage of LLM-based APR through engineering efforts. This can be achieved using two common strategies: (1) quantizing LLM models by converting their weights from high-precision to lower-precision values, and (2) making beam search sequential, where each beam is processed individually through the model before being combined into a single output. However, our findings show that while these methods can reduce memory usage, their memory demands still escalate significantly as beam size increases, leading to substantially high out-of-memory (OOM) rates. Consequently, these engineering efforts alone cannot solve the problem. This realization motivates us to develop a patch generation algorithm that can effectively enhance the performance of LLM-based APR techniques without relying on increasing the beam size and compromising memory efficiency.

In this paper, we introduce FLAMES, a memory-efficient LLM-based APR technique using semantic-guided patch generation. Unlike conventional methods that rely solely on a language model's knowledge and beam search, our approach aims to leverage semantic feedback from test validations to guide LLMs in patch generation. The core idea of FLAMES is to combine LLM-based and search-based APR. This process begins with generating initial patches using greedy decoding, i.e., beam search with a beam size of 1, followed by iterative, semantic-guided searches to refine these solutions. To achieve this, we employ a best-first search algorithm, specifically PG-TD [83], along with semantic feedback from test validations to guide the patch generation of LLMs. This approach offers three key advantages: (1) reduced VRAM consumption through greedy decoding, improving memory efficiency; (2) scalability without increasing VRAM usage while generating more candidate patches; and (3) seamless information exchange between patch generation and validation, enabling efficient exploration of plausible patches.

We conducted an empirical evaluation of our proposed approach, FLAMES, using a dataset comprising 333 bugs from Defects4J [26], 163 bugs from HumanEval-Java [24], and 1,098 bugs from TransformedD4J [41]. We compared FLAMES against 15 state-of-the-art APR techniques and Qwen2.5-Coder-32B, the leading open-source LLM for code in well-known leaderboards [42, 87]. Our experimental results demonstrate that FLAMES can correctly fix 8%, 12%, and 36.5% more bugs than the best baseline on Defects4J, HumanEval-Java, and TransformedD4J, respectively. We also found that FLAMES's semantic-guided patch generation substantially outperforms widely used patch generation strategies in LLM-based APR, including beam search and multiple sampling, with at least 12% and 23% improvements in the number of correctly fixed bugs. Moreover, our analysis of memory efficiency showed that our method could reduce VRAM consumption by 42% to 83%, decreasing peak VRAM requirements from over 80 GB to as low as 12.7 GB across various configurations and models. Despite the improvement in memory efficiency, FLAMES does not compromise time efficiency, even generating plausible patches faster than conventional LLMs using beam search in many cases.

In summary, we have made the following contributions:

- We empirically study the impact of beam size on the effectiveness and memory efficiency of five different LLM-based APR techniques, highlighting the challenges of scaling the search space due to extensive memory consumption of the techniques.
- We introduce a novel approach, namely FLAMES, which fuses LLM-based and search-based APR, utilizing feedbacks from patch validation to efficiently discover plausible candidate patches.
- We empirically evaluate the performance of FLAMES against 15 state-of-the-art baselines and the leading open-source LLM for code. FLAMES correctly fixes 133 bugs in Defects4J, 103 bugs in HumanEval-Java and 456 bugs in TransformedD4J, surpassing the best baseline by 8%, 12% and 36.5%, respectively. FLAMES also outperforms widely-used patch generation strategies by at least 12%. Moreover, FLAMES substantially reduces memory consumption by up to 83% without compromising time efficiency.

2 MOTIVATION STUDY

Beam size is a critical hyperparameter in beam search, the fundamental algorithm for patch generation in LLM-based APR. In this section, we evaluate the impact of beam size on the effectiveness and memory efficiency of LLM-based APR techniques using LLMs. Although prior studies [24, 45, 63, 72] showed that a substantial beam size is needed for optimal APR performance, our experiments reveal that increasing the beam size unduly hinders memory efficiency and subsequently degrades the performance of LLM-based APR, even on state-of-the-art hardware.

RQ_1: How does beam size affect memory efficiency of the LLM-based APR? We measure the use of VRAM and the frequency of out-of-memory crashes in the LLM-based APR techniques with different beam sizes to understand the impact of this parameter on memory efficiency. We selected beam sizes of 10, 25, 50, 100, and

Models	InCoder-1B	InCoder-6B	CodeGen-2B	CodeGen-6B	RepairLLama
Base Model	InCoder [15]		CodeGen-NL [51]		CodeLlama-7B [56]
Pre-training Dataset	InCoder [15]		ThePile [16]		CodeLLama [56]
- Cut-off Date	12.2021		12.2020		08.2023
Fine-tuning Dataset	Recoder [86]				MegaDiff [48]
- Cut-off Date	03.2018				09.2015
Fine-tuning Technique	Full-parameter Fine-tuning				QLORA [10]
Model Size	1B	6B	2B	6B	7B

Table 1: Detailed Information of LLM-based Program Repair techniques used in this study.

200, as these are common in prior works [69, 72, 78, 86]. We do not go beyond a beam size of 200 due to hardware constraints.

RQ_2: How does beam size affect the effectiveness of LLM-based APR? We investigate the impact of beam size on the effectiveness of five LLM-based APR techniques by measuring the number of plausible patches. Similarly to RQ_1, we also selected beam sizes of 10, 25, 50, 100, and 200.

2.1 Experimental Design

2.1.1 Benchmark Dataset. To address these research questions, we conducted experiments with Defects4J, a benchmark of 835 real-world bugs from 17 open-source Java projects [26]. Following prior studies [9, 41, 43, 72, 78], we focused on 333 single-hunk bugs, where patches modify a single contiguous code chunk. This selection aligns with LLM-based APR techniques like InCoder and CodeGen, which are fine-tune for single-hunk bug fixes [22].

2.1.2 Studied LLM-based APR techniques. To select the target techniques for our empirical study, we conducted a literature review and identified suitable techniques based on the following criteria: (1) use beam search as their patch generation strategy, (2) employ LLM with at least one billion parameters, and (3) provide accessible fine-tuned models for Automated Program Repair. Applying these criteria, we identified five LLM-based APR techniques from recent works [22, 57]: CodeGen (2B, 6B) and InCoder (1B, 6B), fine-tuned using full-parameter fine-tuning by Jiang et al.[22]; and RepairLlama (7B), fine-tuned with efficient-parameter fine-tuning, i.e., QLORA, by Silva et al. [57]. Detailed information about these LLMs are provided in Table 1. Since RepairLlama has utilized quantization, we excluded its quantized versions from follow-up experiments.

2.1.3 Memory Reduction techniques. In this study, we also investigate the potential of memory reduction techniques, including quantization and sequential beam search, through engineering efforts and their impact on our findings. **Quantization** is a widely used technique to reduce the memory usage of LLMs by converting their weights from high- to lower-precision values. In this work, we applied 4-bit quantization, reducing the LLM's weights from 32 bits to 4 bits. For clarity, we denote the original LLM-based APR as **full-precision**, and its quantized counterpart as **quantized** in the following sections. Meanwhile, **sequential beam search** is a variant of the implementation of the beam search proposed by Saibo Geng [11], which processes each beam sequentially through the model before concatenating them into a single output.

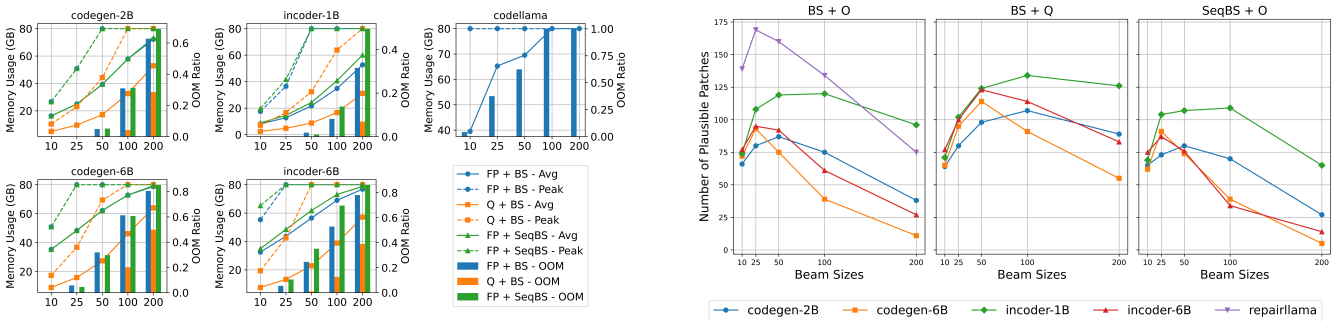
2.1.4 Implementation Details. We implemented LLM-based APR techniques in Python using PyTorch and HuggingFace. To improve reliability and reduce bias, we integrated inference code, repair prompts, and trained models from the original repositories into a unified framework. Our implementations also used HuggingFace's standard beam search. Inference was performed with a batch size of one, processing one buggy program at a time. All experiments were run on a single NVIDIA A100 GPU (80GB VRAM).

2.2 RQ_1: Impact of Beam Size on Memory Efficiency

Figure 1a illustrates the experimental results of RQ_1. Overall, we can see that **VRAM usage increases significantly as the beam size increases from 0 to 200 in the full-precision versions of five studied techniques**. For instance, in InCoder-1B, the average memory usage substantially increases from 20GB at a beam size of 10 to around 60GB at a beam size of 200. Similarly, peak memory usage increases dramatically, reaching maximum memory limits at a beam size of only 50. Notably, this increase in VRAM usage leads to **a substantial increase in crashes due to Out-of-Memory (OOM) errors**. For example, the OOM ratio in InCoder-1B increases from nearly negligible at smaller beam sizes to approximately 50% at a beam size of 200. Larger models, e.g. CodeGen-6B and InCoder-6B, show even higher OOM rates, nearly 80%, respectively.

Next, we investigated 4-bit quantization and sequential beam search, commonly used to optimize model memory requirements, for reducing VRAM usage. As shown in Figure 1a, quantization significantly reduces the average and peak VRAM consumption, thereby reducing OOM crashes. However, sequential beam search presents trade-offs and does not consistently decrease memory usage, as confirmed by its authors and our theoretical analysis (for details, please see our online Appendix [3]). More importantly, **even with these techniques, VRAM usage for LLM-based APR techniques continues to increase with increasing beam size**. For example, CodeGen-2B's average VRAM usage rises from less than 10GB at a beam size of 10 to 60GB at 200, with peak usage nearing the 80GB hardware limit. As a result, despite memory reduction techniques, LLM-based APR techniques still face frequent OOM crashes, occurring in 30–85% of cases with a beam size of 200.

> **Answers to RQ_1:** Both the average and peak memory usage of LLM-based APR techniques increase significantly as the beam size increases, inducing up to 80% out-of-memory crashes on an A100 with 80GB of VRAM. Even with memory reduction techniques, memory usage still escalates, and crash rates remain substantial as the beam size increases.



(a) Memory Usage

(b) Effectiveness

Figure 1: Memory usage and effectiveness of LLM-based APR techniques across beam sizes on an NVIDIA A100 (80GB) using full-precision (FP) and quantized (Q) models. BS and SeqBS denote standard and sequential beam search.

2.3 RQ_2: Impact of Beam Size on Effectiveness

Figure 1b presents our experimental results for RQ_2. Overall, we can see that **the number of plausible patches generated by full-precision versions of LLM-based APR only shows an upward trend at smaller beam sizes and remarkably declines at larger sizes**. For instance, the number of plausible patches generated by RepairLlama increases from 139 at a beam size of 10 to 169 at 25, then drops significantly to 134 at 100 and further to 75 at 200, although the ratio of plausible patches remains at 0.60. This reduction is primarily due to the high frequency of OOM crashes, which range from 30-60% at larger beam sizes, as depicted in Figure 1a. These results highlight the significant impact of LLMs' extensive memory usage on their performance with large beam sizes.

Next, similar to RQ_1, we examined the impact of 4-bit quantization and sequential beam search on memory usage and the effectiveness of LLM-based APR techniques. **Overall, the effectiveness of LLM-based APR techniques remains consistent with sequential beam search, maintaining the same trends. Meanwhile, quantization maintains an upward trend in effectiveness up to a beam size of 100**. For example, the number of plausible patches generated by quantized CodeGen-2B increased from 64 at a beam size of 10 to 107 at a beam size of 100, surpassing the original full-precision model by 42.7% at the same beam size. This improvement is due to a reduction in OOM crashes, from 30% to less than 5%. However, despite these gains, the effectiveness of quantized models still declines at larger beam sizes, such as 200, similar to the trend seen in the full-precision models. For example, the effectiveness of the quantized CodeGen-2B dropped from 114 plausible patches at a beam size of 50 to 91 at 100, and further to 55 at 200, reflecting declines of 20% and 48%, respectively.

> **Answers to RQ_2:** As the beam size increases, the effectiveness of LLM-based APR increases, peaking at mid-range beam sizes before declining at larger sizes due to higher OOM crash rates, particularly in larger models. Even with memory reduction techniques, the APR's effectiveness still quickly degrades at beam sizes larger than 100.

3 APPROACH

Our findings in Section 2 highlight that scaling the beam size to a certain threshold may help improve the effectiveness of APR at the cost of extensive memory consumption, above which the effectiveness quickly degrades even on state-of-the-art hardware. To address these limitations, we propose a novel approach, FLAMES, which combines LLM-based and search-based APR for memory-efficient, semantic-guided patch generation.

Figure 2 illustrates the overall pipeline of FLAMES, which employs PG-TD as the core algorithm to guide patch generation in LLM-based APR. PG-TD is a best-first search algorithm, inspired by Monte Carlo Tree Search [33], to decode Transformer models. PG-TD formulates the decoding process of Transformers models as a search problem on a tree structure whose nodes represent tokens. PG-TD then iteratively expands this tree with the information provided by the models and searches for an optimal candidate using a predefined reward function. In the following sections, we first present a high-level overview of patch generation of LLM-based APR with PG-TD and then present a detailed methodology of PG-TD's step in APR context.

3.1 Patch Generation with PG-TD

Algorithm 1 outlines the patch generation process in APR using the PG-TD algorithm. At a high level, PG-TD conducts an iterative search for plausible patches. This search proceeds by refining partial repair candidates over multiple iterations. At each iteration, PG-TD selects the most promising partial patch from the current search space, as described in Section 3.2. The chosen partial patch is then used in two key operations. First, since it is the most promising candidate, PG-TD will further expand the search space by incorporating most of the next potential tokens for the partial patch, predicted by an LLM-based APR model (Section 3.3). Second, the partial patch is simulated into a complete patch candidate using the same LLM-based APR model and is subsequently evaluated against a predefined set of correctness specifications to compute a reward value (Section 3.4). This reward is then backpropagated through the search tree to update the estimated potential of the nodes in the partial patch (Section 3.5). This iterative process continues until

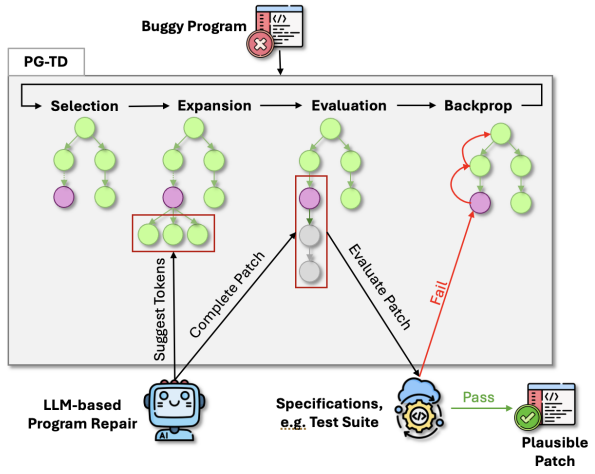


Figure 2: Overview of FLAMES

a plausible patch is discovered, that is, a candidate that satisfies the given specifications or the iteration limit is reached.

3.2 Selection

At this stage, our goal is to select the most promising partial patch to explore. To achieve this, we employ a policy algorithm that strikes a balance between exploiting potentially high-reward states and exploring less-visited ones. Specifically, an action (i.e., adding a token) is chosen based on three criteria: (1) the action's historical reward in relation to the current state is high, indicating strong potential; (2) language models (LLMs) predict that the action is a suitable next token, as reflected by its prediction probability; and (3) the resulting state after taking the action is under-explored.

To implement this approach, we evaluate three commonly used policy algorithms in Monte Carlo Tree Search (MCTS): UCB [33], and two variants of P-UCB [58], one with fixed weights (Fixed P-UCB) and the other with variable weights (Variable P-UCB). Our results indicate that Variable P-UCB is the most effective algorithm for the APR task, leading us to adopt it as our primary policy algorithm (See Section 4.6 for further details.)

3.3 Expansion

This stage aims to expand the search tree by adding possible next tokens. As the number of possible tokens in the vocabulary is huge, random sampling, which is used in MCTS [33] is likely to return in invalid and low-potential tokens. Therefore, FLAMES leverages fine-tuned LLM to suggest top-k next tokens with the highest prediction score given the current state following PG-TD [83]. Note that, as a state can be re-visited multiple times during the tree search, FLAMES also leverages a caching mechanism to avoid redundant computation. In particular, whenever LLMs make a prediction, the input sequence, i.e., state and top-k next tokens, is cached using a hash map. Then, when the algorithm needs LLM to predict top-k next tokens, such information can be reused if it exists in the cache.

Algorithm 1 Semantic-guided Patch Generation with PG-TD. *SELECT*, *EXPAND* and *BACKPROP* are presented in Section 3.2, 3.3 and 3.5, respectively. *SIMULATE* and *EVAL* are presented in Section 3.4.

Require: bp: buggy program; $\mathcal{L}$: LLM-based Program Repair model; $\mathcal{S}$: set of correctness specifications; max_iters: number of iterative refinement steps
Ensure: cp: plausible patch for bp **or** *None* if no patch found

```
root ← L.initial_token                          ▷ Initialize search tree
reward ← {}                                     ▷ Initialize reward dictionary
for i = 1 to max_iters do
    pp ← SELECT(root, reward)                   ▷ Select best partial patch
    root ← EXPAND(root, L, bp)                  ▷ Expand search tree
    cp ← SIMULATE(pp, L, bp)                    ▷ Simulate complete patch cp
    r ← EVAL(cp, S, bp)                         ▷ Evaluate rewards of pp using cp
    if r ≠ 1 then
        reward ← BACKPROP(reward, pp, r)        ▷ Update rewards
    else
        return cp                               ▷ Return plausible complete patch
    end if
end for
return None
```

In this work, we set $k = 10$ and further discuss the impact of k on our method in Sections 4.6.2.

3.4 Evaluation

This stage aims to evaluate the potential of the current state, i.e., how likely the partial patch leads to a optimal (complete) patch. Similar to expansion, it is impossible to evaluate the current state using an random simulation as used in MCTS algorithm. Consequently, FLAMES leverages beam search to automatically complete the current state, i.e., a partial patch pp, to form a complete patch cp. However, it is noteworthy that leveraging the beam search as in original PG-TD still hinders memory efficiency. Therefore, instead of beam search, FLAMES utilizes a greedy search that is equivalent to a beam search with a beam of size 1, to find the complete patch for the current state while keeping low memory consumption.

Then, given the completed patch, FLAMES estimates the potential of current state by calculating a reward function $R(cp, spec)$. Theoretically, $spec$ and reward function R can takes any kinds of specification and a corresponding pre-defined reward function. In this work, following common practices in literature [17], we leverage developer-written test cases as our specification and utilize the passing ratio, which is widely-used in search based APR [17], as our reward function. More formally, given a program cp and a set of test cases $spec$, we define our reward function as follows: $R(cp, spec) = \frac{f_{pass}}{f_{pass}+f_{fail}}$ with f_{pass} and f_{fail} is the number of passing and failing test cases observed when running cp on $spec$.

3.5 Backpropagation

This stage aims to update the value of nodes in the tree for guiding next steps of the search process. Intuitively, the stage allows

FLAMES to provide feedback from test validation for guiding the patch generation process of LLMs. Particularly, the reward of a state (calculated based on automatically-completed program) is backpropagated to its parents recursively until the root is reached. Similar to other reinforcement learning algorithms [27, 60, 65], we aim to maintain the maximum observed reward as an expected return value for a given state s and the corresponding action a. To do so, for all state and action pair (s, a) along the path from the current state to the root node, FLAMES updates their Q-value by $Q(s, a) = max(Q(s, a), R(cp, spec))$, where $R(cp, spec)$ is the reward value of the current state action pair (s, a) with cp is an automatically complete patch of s.

3.6 Implementation Details.

We implemented FLAMES by extending the PG-TD [83] framework based on the DynaGym toolkit. We use RepairLLama, the best-performing model among LLM-based repair techniques, as our default LLM-based repair model. For repair, FLAMES generates up to 200 patches and sets a 30-minute timeout for each bug, as recommended by developers [52]. Experiments are conducted on an NVIDIA A100 GPU with 80 GB VRAM, 250 GB RAM, and a 32-core Intel Xeon CPU at 2.90 GHz, running Red Hat Enterprise Linux 9.3 and Java 1.8.0_241. This setup is comparable to those used in related works [57, 63].

4 EMPIRICAL EVALUATION

4.1 Research Questions

Continuing from our motivation study in Section 2, our empirical evaluation aims to answer the following four research questions.

RQ_3: How effective is FLAMES? We assess the effectiveness of FLAMES on the Defects4J [26], HumanEval-Java [22] and TransformedD4J [41] datasets, comparing our proposed technique to 15 state-of-the-art APR techniques and Qwen2.5-Coder-32B, the best open-source LLM for code in well-known leaderboards [42, 87].

RQ_4: How efficient is FLAMES? We evaluate the time and memory efficiency of FLAMES and compare it with standard and sequential beam search algorithms.

RQ_5: How effective is FLAMES's semantic-guided patch generation? We investigate the effectiveness of FLAMES's semantic-guided patch generation by implementing our approach with six LLMs for APR and comparing their effectiveness to the full-precision models using two widely used patch generation strategies: beam search [57, 78] and multiple sampling [63, 68].

RQ_6: How do different configurations affect the performance of FLAMES? Finally, we examine the impact of various key factors on the effectiveness of FLAMES, including the reward function, the expansion size, and the policy algorithm.

4.2 Experimental Setup

4.2.1 Benchmark Dataset. Similar to our initial motivation study (Section 2.1), we evaluated our method using 333 single-hunk bugs from the Defects4J dataset [26]. Furthermore, we used HumanEval-Java [22] and TransformedD4J [41], which contain artificial bugs introduced by applying bug injection to normal programs and semantic-preserving transformations to Defects4J bugs, respectively, to mitigate the risk of data leakage.

4.2.2 APR Baselines. To evaluate the effectiveness of our approach (RQ3), we compared FLAMES with the leading baselines in various APR categories: Template-based, NMT-based, Cloze-based, and LLM-based APR techniques. For LLM-based APR, we evaluated five models using their best variants identified in RQ_2. For Cloze-based APR, we included FitRepair [68], AlphaRepair [72], and Repilot [63]. We also compared FLAMES with NMT-based methods, i.e., ITER [79], Recoder [86], KNOD [23], and RewardRepair [78], as well as template-based approaches, i.e., GAMMA [82], TENURE [47], and TBar [43]. Finally, we also included Qwen2.5-Coder-32B [21], the leading LLM for code on well-known leaderboards [42, 87]. Following standard practices [9, 22, 24, 63, 72], we performed evaluations under the assumption of perfect fault localization, which ensured that variations in fault localization techniques do not affect the results. For LLM-based techniques, we used the same configurations as in Section 3.6. For other techniques, we collected bug fix results from their original papers and replication packages, following common practices in prior works [24, 63, 72, 78].

4.2.3 Patch Generation Algorithm Baselines. To evaluate the effectiveness of FLAMES's semantic-guided patch generation (RQ5), we also compared our approach with two well-known patch generation strategies: beam search and multiple sampling. For beam search, we leverage the standard implementations from the Huggingface library, a widely adopted toolset for training and deploying LLMs. For multiple sampling, we first set the LLM temperature to 1 to maximize its diversity. Then, we repeat the default patch generation of these models (using beam search) until we reach 200 patches.

4.2.4 Evaluation Metrics and Patch Correctness Assessment. We assessed the effectiveness of APR techniques using a standard metric: the number of correct patches. To identify patch correctness, following prior works [34, 41, 45, 72, 78], we conducted a manual evaluation to determine if they were syntactically or semantically equivalent to developer-written patches. Note that while manual patch correctness assessment is precise, it is resource-intensive and requires considerable human effort.

4.3 RQ_3: Comparison with Existing Techniques

4.3.1 Repair Success. Table 2 presents the number of correct fixes generated by our approach, FLAMES, and baseline methods in Defects4J. FLAMES successfully repairs 133 of the 333 single-hunk bugs, outperforming all existing APR techniques.

Comparison with RepairLLama's Variants. Specifically, our approach generates correct patches for 21 and 43 more bugs than two other variants with beam search and multiple sampling of its core LLM, RepairLLama. These results highlight the advantages of our semantic-guided patch generation approach using PG-TD. This method enables FLAMES to scale to larger beam sizes without compromising memory efficiency, as observed in RepairLLama with beam search. Meanwhile, it also guides the patch generation process with semantic feedback, resulting in a more effective approach than multiple sampling without guidance. Although this approach slightly reduces the precision from 0.66 to 0.63, this trade-off is justified by a 19% increase in repair success. Moreover, FLAMES

Table 2: Comparison with state-of-the-art APR techniques on Defects4J benchmark. #Correct and #Plausible denotes the number of correct and plausible patches generated by each tool. RepairLLama (+BS) and (+MS) denotes the results of RepairLLama with beam search and multiple sampling, respectively.

Techniques	#Correct	#Plausible	Precision
TBar	46	-	-
GAMMA	87	-	-
TENURE	84	-	-
RewardRepair	79	-	-
Recoder	43	-	-
KNOD	110	-	-
ITER	55	-	-
AlphaRepair	95	-	-
FitRepair	123	-	-
Repilot	116	-	-
InCoder-1B	42	75	0.56
InCoder-6B	50	96	0.52
CodeGen-2B	41	88	0.47
CodeGen-6B	49	94	0.52
Qwen2.5-Coder-32B	68	112	0.61
RepairLLama (+BS)	112	170	0.66
RepairLLama (+MS)	90	151	0.59
FLAMES	133	211	0.63

maintains a higher precision than other LLM-based APR techniques and RepairLLama with multiple sampling.

Comparison with State-of-the-Art Techniques. Additionally, FLAMES significantly outperforms other state-of-the-art APR baselines. It correctly fixes 10 more bugs than FitRepair, the best baseline, achieving an 8% improvement. Additionally, FLAMES outperforms the top NMT-based technique, KNOD, and the top template-based approach, GAMMA, by 21% and 53%, respectively, in the number of correct patches. Notably, FLAMES achieves this performance with a budget of only 200 validated patches and a 30-minute pause per bug, while FitRepair and KNOD require 1,000–5,000 validated patches and a pause of 5-hours per bug. Furthermore, FLAMES outperforms Qwen2.5-Coder-32B, the leading LLM for code, by 51%. This advantage comes from two key factors: first, this model is not specifically trained for APR, unlike our core model, RepairLlama; second, FLAMES incorporates semantic-guided patch generation that enhances its repair effectiveness.

Unique Fixes. We found that FLAMES achieves the highest number of unique fixes among the best baselines for NMT-based (KNOD [23]), Cloze-based (FitRepair [68]) and LLM-based (RepairLLama [57]) APR approaches. Particularly, RepairLlama, KNOD, and FitRepair contribute 6, 5, and 9 unique fixes, respectively, while FLAMES can fix 14 unique bugs. Moreover, there are 85 correct fixes that are not addressed by these four leading techniques. By combining all the techniques, we can successfully fix 299 bugs. This shows that FLAMES can be complementarily used with existing APR methods such as KNOD and FitRepair to significantly increase the number of correct fixes generated.

Table 3: Comparison with RepairLLama on multi-location bugs from Defects4J.

Techniques	Correct/Plausible Patches
RepairLLama	16/35
FLAMES	24/59

Table 4: Comparison with state-of-the-art APR techniques on HumanEval-Java and TransformedD4J benchmark. The results are displayed as x/y, with x, y denotes the number of bugs with correct and plausible patches, respectively.

Techniques	HumanEval-Java	TransformedD4J
CodeGen-6B	57/65	_/393
InCoder-6B	71/81	_/427
Qwen2.5-Coder-32B	68/111	_/403
RepairLLama	92/118	334/581
FLAMES	103/133	456/776

4.3.2 Repair Success on Multi-Location Bugs. Beyond the comparison on single-location bugs, we also conduct experiments on multi-location bugs following the settings of RepairLLama [57] with 172 bugs from Defects4J. In this setting, each bug requires edits on multiple non-consecutive locations within a single function. We compare the effectiveness of FLAMES with our base model and also the best LLM baseline, RepairLlama to further understand the impact of FLAMES on repairing multi-location bugs. The detailed results are presented in Table 3

Our experimental results show that while RepairLLama (the best LLM baseline) generates 35 plausible patches for these bugs, FLAMES (with RepairLLama as the base LLM) can generate 59 plausible patches, demonstrating 68% improvement over the best baseline. Further inspection reveals that FLAMES generates 24 correct patches out of 59 plausible ones, compared to only 16 correct patches generated by RepairLLama. This result confirms the advantages of our patch generation for such complex scenarios.

4.3.3 Generalizability. To address concerns related to data leakage [22] and benchmark overfitting [12], we extended our evaluation of FLAMES and the four most effective LLM-based APR techniques from the previous experiment. Specifically, we evaluated their performance on the HumanEval-Java [24] and TransformedD4J [41] benchmarks. Given the large number of plausible patches generated by these techniques on TransformedD4J, we limited our correctness assessment to patches produced by the two most effective methods, namely FLAMES and RepairLLama.

Overall, FLAMES significantly outperforms the baselines in terms of the number of correct patches. For example, in HumanEval-Java, FLAMES successfully fixes 11 more bugs than the best-performing baseline, RepairLlama, which generates 92 correct fixes, representing an improvement of 12%. Similarly, in TransformedD4J, FLAMES achieves a 36.5% improvement over the best baseline. These results demonstrate the generalizability of FLAMES across various evaluation benchmarks, highlighting its effectiveness in generating correct fixes in different data sets.

Table 5: Comparison of memory usage among the full-precision LLM (Full-Precision), quantized LLM (Quantized), full-precision LLM with sequential beam search (SeqBS) and full-precision LLM with FLAMES.

Models	Method	Average (GB)	Peak (GB)	OOM (%)
CodeGen-2B	Full-Precision	72.2	80	62.8
	Quantized	52.9	80	28.5
	SeqBS	68.2	80	69.1
	PG-TD	**12.1**	**13.9**	**0.0**
CodeGen-6B	Full-Precision	78.7	80	80.5
	Quantized	64.0	80	49.8
	SeqBS	75.5	80	85.3
	PG-TD	**27.1**	**31.7**	**0.0**
InCoder-1B	Full-Precision	52.7	80	31.5
	Quantized	31.2	80	6.9
	SeqBS	56.1	80	49.5
	PG-TD	**7.4**	**12.7**	**0.0**
InCoder-6B	Full-Precision	76.8	80	78.1
	Quantized	57.2	80	39.0
	SeqBS	75.6	80	86.6
	PG-TD	**32.4**	**37.4**	**0.0**
RepairLlama	Full-Precision	66.6	80	62.8
	PG-TD	**8.1**	**22.8**	**0.0**

Answers to RQ$_3$ (Effectiveness): FLAMES successfully fixes 133, 103, and 456 bugs on Defects4J, HumanEval-Java, and TransformedD4J, significantly outperforming the best baselines by 8%, 12% and 36. 5%, respectively.

4.4 RQ$_4$: Efficiency

4.4.1 Memory efficiency. Table 5 compares the memory usage of FLAMES with three baselines: (1) the full-precision LLM with beam search, (2) the quantized LLM with beam search, and (3) the full-precision LLM with sequential beam search, across five different LLMs. We evaluated VRAM usage using three key metrics: average VRAM usage, peak VRAM usage observed in our dataset, and out-of-memory ratios. Since our hardware is capped at 80GB VRAM, we assign a memory usage of 80GB to data points that experience OOM errors. Consequently, in cases of OOM, the reported memory usage represents a lower bound of the actual values.

Overall, the VRAM usage of FLAMES demonstrates a substantial reduction across all evaluated models, outperforming both the original full-precision configurations and the two widely used memory reduction techniques: Quantization and Sequential Beam Search (SeqBS). For example, in the case of the InCoder-1B model, the average VRAM usage drops from 52.7 GB in the original configuration to just 7.4 GB with FLAMES, representing a reduction of 86%. In comparison, the quantization approach reduces memory usage to 31.2 GB, while SeqBS increases it to 56.1 GB, making FLAMES the most memory-efficient method.

Table 6: Comparison of average running times of FLAMES and beam search (BS) for finding plausible patches with statistical measures on different models and number of generated patches (# Patches)

Model	# Patches	Avg. Time (s)		$\|\delta\|$	p-value
		Flames	BS		
CodeGen-2B	10	**37.99**	49.43	0.33	0.002
	25	**55.89**	63.12	0.22	0.013
	50	**68.66**	88.07	0.24	0.004
	100	**92.63**	124.72	0.27	0.005
	200	**93.88**	133.40	0.37	0.006
InCoder-1B	10	**48.18**	68.71	0.35	0.001
	25	91.39	**86.82**	0.07	0.195
	50	**115.79**	138.61	0.11	0.081
	100	**129.62**	177.19	0.12	0.059
	200	**161.58**	199.17	0.08	0.178
RepairLLama	10	**44.40**	57.55	0.25	0.001
	25	90.95	**79.21**	0.13	0.027
	50	103.79	**96.81**	0.15	0.014
	100	126.74	**111.42**	0.15	0.022
	200	**98.42**	101.35	0.22	0.018

Regarding peak VRAM usage, while baseline methods frequently reach the 80 GB VRAM limit, FLAMES maintains peak values well below this threshold. Notably, FLAMES achieves a maximum VRAM usage of just 12.7 GB for the smallest model (InCoder-1B) and 37.4 GB for the largest model (InCoder-6B). In contrast, both the original full-precision and SeqBS configurations consistently hit the 80 GB VRAM limits, increasing the risk of out-of-memory (OOM) errors.

Furthermore, in terms of OOM ratios, FLAMES completely eliminates OOM errors across all models, achieving a 0% OOM ratio, while the original full-precision and SeqBS methods frequently experience severe memory failures. For example, the full-precision CodeGen-6B model encounters an 80.5% OOM rate, making it **unusable** in memory-constrained environments. Even quantization, while reducing memory usage, does not guarantee the elimination of OOM errors, as observed with CodeGen-6B, which still reports an OOM ratio of 49.8%.

Answers to RQ$_{4.1}$ (Memory Efficiency): FLAMES can significantly reduce VRAM usage in LLMs, successfully minimizing average VRAM usage by up to 83% and reducing peak VRAM requirements from 80 GB to as low as 12.7 GB across various configurations and models. Additionally, FLAMES eliminates OOM errors, while the baselines witness OOMs in up to 86.6% of the data points.

4.4.2 Time efficiency. In this experiment, we evaluated the time efficiency of LLM-based APR using our proposed method, FLAMES, compared to traditional beam search techniques. Our aim is to validate the hypothesis that "*FLAMES can identify plausible patches more quickly than the traditional beam search algorithm*". To validate this hypothesis, we measure the time each method takes to successfully identify plausible patches, employing the Mann-Whitney-Wilcoxon (MWW) statistical test and Cliff's Delta for effect size

assessment. This experiment focuses on the CodeGen-2B, InCoder-1B, and RepairLlama models, chosen for their lower incidence of out-of-memory (OOM) crashes, ensuring a robust sample size for statistical testing. The detailed results are presented in Table 6.

For the CodeGen-2B model, our hypothesis is supported in all examined numbers of generated patches, with p values between 0.002 and 0.013 and effect sizes between 0.22 and 0.37, indicating that FLAMES is more time efficient than beam search. Notably, FLAMES reduce time by approximately 11.44 seconds and by 39.52 seconds with 10 and 200 generated patches.

The RepairLlama model also supports the hypothesis with p-values between 0.001 and 0.027 across all examined numbers of generated patches, confirming the time efficiency of FLAMES over the beam search, although the effect sizes range from negligible to small. Consequently, FLAMES shows lower average running times with 10 and 200 generated patches. For other configurations, while FLAMES generally performs faster than the beam search, the occasional longer run times of FLAMES skew the average, leading to variability in the results.

In contrast, the InCoder-1B model exhibits greater variability between the two methods. Although FLAMES tends to generate patches faster at higher numbers of generated patches (50, 100, and 200), it performs worse than BS with 25 generated patches. The most favorable performance for FLAMES is observed with 10 generated patches, where it significantly outperforms the beam search (p-value 0.001). However, for 25 and 200 generated patches, the differences in running times lack statistical significance, indicating that the benefits of FLAMES do not apply uniformly in all configurations within this model.

Answers to $RQ_{4.2}$ (Time Efficiency): FLAMES significantly improve time efficiency in LLMs, surpassing traditional beam search methods.

4.5 RQ_5: Effectiveness of Semantic Guided Patch Generation

In this experiment, we evaluated the effectiveness of FLAMES's semantic-guided patch generation in comparison to beam search and multiple sampling. These methods were evaluated in five LLM-based automated program repair (APR) models, as introduced in Section 2.1.2, together with Qwen2.5-Coder-32B, a state-of-the-art LLM for code. The detailed results are presented in Table 7.

FLAMES consistently outperforms beam search in all models, with improvement rates ranging from 12% to 48% in terms of the number of correct patches. For example, FLAMES generates 21 more correct patches than RepairLlama, our default base model. This is due to FLAMES 's effective use of GPU resources which allows for a more thorough exploration of the large patch space. In contrast, beam search requires substantial VRAM, resulting in a high out-of-memory crash rate, as discussed in RQ_4 and RQ_1.

A common approach to mitigate memory constraints is multiple sampling, which involves performing beam search with a low beam size and subsequently sampling the model multiple times to generate additional candidate patches. This method enables the generation of more candidate patches while reducing memory consumption. However, our findings indicate that this approach results in lower effectiveness compared to beam search. For instance, in the case of RepairLLama, multiple sampling leads to an approximately 20% decline in performance. Consequently, FLAMES significantly outperforms multiple sampling across models, achieving improvements ranging from 23% to 68%. We hypothesize that this performance gap arises because FLAMES leverages feedback from test validation to guide patch generation, whereas multiple sampling relies solely on random sampling from the model's output.

Table 7: The number of plausible patches generated by FLAMES's patch generation algorithm compared to beam search (BS) and multiple sampling (MS) on different LLMs

LLM	BS	MS	FLAMES
CodeGen-2B	41	38	**57**
InCoder-1B	57	52	**64**
RepairLLama	112	90	**133**
CodeGen-6B	49	39	**70**
InCoder-6B	50	44	**74**
Qwen2.5-Coder-32B	_	68	**91**

In conclusion, consistent performance improvements across various models strongly support the effectiveness of FLAMES's semantic-guided patch generation compared to widely used patch generation approaches such as beam search and multiple sampling.

Answers to RQ_5 (Effectiveness of Patch Generation): FLAMES's semantic-guided patch generation demonstrates superior performance across multiple LLMs in APR tasks, consistently outperforming widely used patch generation methods: beam search and multiple sampling, by at least 12% and 23%, respectively.

4.6 RQ_6: Ablation study

To systematically assess how each component influences the performance of FLAMES, we investigated three key factors: the semantic-guided reward function (for the evaluation phase), the size of the expansion k (for the expansion phase), and the choice of the policy algorithm (for the selection phase).

4.6.1 Reward Function. In this experiment, we evaluate the effectiveness of our reward function, as detailed in Section 3.4. We hypothesize that this function will guide the generation of more plausible patches by leveraging the ratio of passing test cases. To test this hypothesis, we compare our approach with a variant that depends solely on the models' probability outputs and disregards feedback from our semantic reward function. Our experimental results show that incorporating a semantic reward function can lead to a 40% increase, from 150 to 210, in plausible patches. This substantial improvement highlights the value of semantic guidance in refining the search process to concentrate on patches that are more likely to succeed.

4.6.2 Expansion Size (k). Next, we explore how the expansion size k affects FLAMES by experimenting with various values of k,

including 3, 5, 7, 10, 15, 20, 25 and 30. Our findings show that as k increased from 3 to 10, the number of plausible patches increased from 190 to 210. Unfortunately, a further increase of k to up to 30 showed minimal gains, with plausible patches ranging from 202 to 204. This is due to the broader, yet shallower search at higher k, which favors exploration over the exploitation needed for certain bugs. This trend suggests that a larger k may allow for a more thorough exploration of the solution space, enhancing the likelihood of identifying high-quality patches. However, this comes at a trade-off between the depth of exploration and exploitation.

4.6.3 Policy Algorithms. Finally, we investigated the impact of different policy algorithms on the generation of plausible patches. We examined the performance enhancement observed with advanced policy algorithms, including the original UCB [33], and two variants of P-UCB[58]—one with fixed weights (Fixed P-UCB) and one with variable weights (Variable P-UCB). Our experimental results show that Variable P-UCB is the most effective algorithm, generating 210 plausible patches. This policy algorithm outperforms UCB and Fixed P-UCB by 17% (179 plausible patches) and 11% (188 plausible patches), respectively. The performance of Variable P-UCB likely stems from its ability to dynamically adjust the balance between exploration and exploitation during the search process, thereby optimizing the discovery of plausible patches.

5 DISCUSSION

5.1 Limitations

First, while our approach, FLAMES, correctly fixes 21 more bugs and 42 unique bugs compared to the original RepairLlama, there are 21 bugs that the original model could repair but FLAMES could not. In an in-depth analysis on the 21 bugs that FLAMES cannot repair, we observed that the frequent pattern that these bugs most often require entirely new code segments. However, we also notice that these patterns appear in bugs that can be successfully repaired by FLAMES but cannot be repaired by RepairLlama. Therefore, it does not show a strong separation between the patterns of bugs fixed and not fixed by FLAMES. Therefore, we hypothesize that the failures stem from FLAMES's trade-off strategy rather than specific types of bug. Specifically, FLAMES's failure to repair these bugs can possibly be attributed to FLAMES's smaller expansion size of 10. This limitation compromises FLAMES's exploration, resulting in a less comprehensive search space than that of RepairLlama's beam search, which considers up to 25 possible tokens of expansions. Consequently, FLAMES misses some bugs that could potentially be fixed with a broader search scope. Although a straightforward solution would be to increase the expansion size k, doing so introduces a trade-off between exploitation and exploration, as shown in Section 4.6.2. Therefore, further analysis is essential to optimize the balance between expansion size and efficiency to enhance FLAMES's performance.

Second, FLAMES is constrained by the design of its reward function, which relies on the number of passing and failing test cases. While our experiements demonstrated the effectiveness of our reward function on guiding LLM-based APR to generate more plausible patches and thereby more correct patches. It is important to note that a substantial increase in plausible patches does not always result in a proportionally large increase in correct patches. Moreover, it also raise the problem of test overfitting as observed in search-based APR [38, 55]. We believe that refining the reward function with additional criteria such as naturalness [28, 70], syntax/semantic similarity [64, 76] or history versions [36, 50, 61] could enhance both the effectiveness and precision of FLAMES.

5.2 Threats to validity

5.2.1 Internal Validity. A main threat to internal validity stems from the issue of data leakage, where evaluation data might overlap with training and fine-tuning data. To mitigate this risk, we followed prior studies [22, 57] and utilized the HumanEval-Java dataset for evaluation. This dataset consists of 163 bugs introduced by injecting faults into the HumanEval benchmark [8]. Additionally, we employed TransformedD4J, which comprises 1,090 bugs generated through semantic-preserving transformations of Defects4J, further reducing the likelihood of data leakage. Note that, HumanEval-Java and TransformedD4J datasets were publicly released in July 2023 and January 2024, respectively. In contrast, CodeLlama, the base LLM underlying RepairLlama, was released in August 2024. This temporal gap significantly reduces the likelihood of data leakage during the pre-training phase. Furthermore, the fine-tuning data used in the development of RepairLlama was extracted from the Boa dataset [13] collected in September 2015, further minimizing the possibility of overlap with our evaluation datasets. Therefore, we believe that the risk of data leakage in our study is negligible. Another concern is the manual evaluation of patch correctness. To address this, authors and an external annotator rigorously analyze each patch to confirm semantic equivalence to developer-written patches. Moreover, to ensure transparency, we also have made all patches generated by NPR techniques and our assessment results available in our replication package.

5.2.2 External Validity. Our research shares a common threat to external validity with prior works [24, 41, 57, 63, 71, 72] that our results may not generalize across different programming languages and datasets. To mitigate this risk, we evaluated our model, FLAMES, and state-of-the-art baselines using three datasets, Defects4J [26], HumanEval-Java [22] and TransformedD4J [41], and found our performance generalize across these datasets. Additionally, the core contributions of our approach are language-agnostic, suggesting potential applicability to various programming languages. Therefore, we believe that this risk is minimal.

Another external validity relates to our generalization to larger LLMs such as ChatGPT or Claude models. FLAMES modify the LLM decoding strategy and thus require access to LLMs. This avoids evaluation on such closed-source LLMs. As an alternative, we have done our best to conduct experiments on CodeQwen2.5, the best open source code LLM across leaderboards, and demonstrate the effectiveness of FLAMES on this model in Table 7. Moreover, our experiments for RQ5 demonstrate the generalizability of FLAMES across six well-known LLMs with diverse model architectures and sizes. Therefore, we believe that FLAMES is capable of generalizing effectively to different LLMs.

5.2.3 Construct Validity. Our methodology may encounter threats to construct validity concerning the appropriateness of our evaluation metrics. Due to limited human resources, besides the number of correct patches, we leverage the number of plausible patches as a proxy effectiveness metric, which might not fully capture the actual effectiveness of APR techniques. To minimize this risk, we only utilized this metric when comparing different variants of the same APR techniques, where we observed consistent trends between correct patches and plausible patches.

In addition, we also share a common threat to construct validity with previous work [9, 22, 24, 63, 72] on the assumptions of perfect fault localization. However, we did our best efforts to ensure a fair comparison between FLAMES and LLM-based Program Repair by adopting the same fault localization assumptions as our base models as we reused their preprocessing pipelines. For example, in the default setting, our base model, RepairLlama, requires perfect fault localization, so that FLAMES also needs to use perfect fault localization. However, for multi-location bugs, RepairLlama only needs a coarse-grained bug region. As shown in Section 4.3.2, FLAMES can effectively repair multi-location bugs under this assumption. These results suggest that FLAMES may operate without perfect fault localization, provided that its base models support such flexibility.

Finally, as FLAMES relies on test case, it is possible that its performance may not generalize to lower-quality test suite. To address this concern, we evaluated the impact of test suite quality on FLAMES by grouping bugs according to line coverage of their test suites: <60%, 60–80%, and >80%. FLAMES produced plausible patches for 65%, 69%, and 62% of bugs in these groups, and correct patches for 40%, 42%, and 40%, respectively. However, the precision for the >80% coverage group was slightly higher at 0.64, compared to 0.61 and 0.62 in the lower coverage groups. These results suggest that FLAMES performs consistently across varying test suite quality, with a slight decrease in precision on lower coverage suites.

6 RELATED WORKS

6.1 Large Language Models for Automated Program Repair

Recent advancements in Large Language Models (LLMs), pre-trained on extensive datasets, have significantly advanced various coding tasks [14], such as code generation [8, 15], understanding [1, 49], and analysis [29, 39]. In the field of APR, LLMs have also shown great promise when employed through either fine-tuning techniques [20, 22, 57] or prompting methods [71, 73].

Fine-tuning approaches concentrate on refining the weights of code-specific LLMs like CodeLLama [56] and InCoder [15], using APR datasets including MegaDiff [48]. Notably, Jiang et al.[22] demonstrated significant enhancements in LLMs for APR through full-parameter fine-tuning, showing remarkable improvements over the original models. Silver et al.[57] explored the effectiveness of the QLORA [10] fine-tuning method, assessing various input and output formats in LLM-based APR and introducing RepairLLama, an advanced APR technique using CodeLLama. Jin et al.[25] introduced InferFix for fixing security vulnerabilities by fine-tuning Codex model and retrieval augmented generation.

Prompting approaches aim to directly leverage LLMs without finetuning. Particularly, they mainly focus on designing effective prompting which provide feedbacks to guide LLMs to generate correct programs. Notably, Xia et al. [73] propose conversation-driven APR, namely ChatRepair, which guide ChatGPT with instant feedbacks such as test name, relevant test code, and error message, to perform APR in a conversational style. Ahmed et al. [2] and Yin et al. [80] levarages chain-of-thought prompting to improve the effectiveness of LLMs on APR. Zhang et al.[84] and Bouzenia et al.[5] introduce AutoCodeRover and RepairAgent,tools that enable LLMs to follow human-like debugging steps using multi-agent systems to fix real-world issues.

Different from these studies that typically employ beam search to generate candidate patches, our work empirically investigates the limitations of beam search, particularly when implemented on standard hardware setups. Motivated by these shortcomings, we propose the use of PG-TD to guide LLM-based APR.

6.2 Efficiency of Automated Program Repair

In addition to evaluating effectiveness, several studies also focus on the efficiency of APR [6, 7, 44, 54, 75]. Liu et al. [44] conducted a systematic assessment of 16 APR techniques, revealing inefficiencies in state-of-the-art APR tools and advocating for further industry exploration of efficiency impacts. Chen et al. [6] introduced UniAPR, a patch validation framework that enhances efficiency by avoiding unnecessary restarts of the JVM for all existing bytecode and source code-level APR techniques. Xiao et al. [74] developed ExpressAPR, which accelerates APR by integrating five mechanisms: mutant schemata, mutant deduplication, test virtualization, test prioritization, and parallelization. Chen et al. [7] proposed utilizing XGBoost to develop on-the-fly prioritization techniques to expedite APR. Kim et al. [30] and Gresino [31] presented novel patch-scheduling algorithms for enhancing APR time efficiency.

Different from these studies that typically focus on time efficiency, our approach focus on memory efficiency, motivated by the extensive memory requirements of LLMs. Our work is closely related RepairLLama [57], which explores efficient fine-tuning of LLMs during the training phase. Different from this work, we investigate memory efficiency during the patch generation phase.

7 CONCLUSION

In this study, we empirically evaluated the impact of beam size on the memory efficiency and effectiveness of LLM-based APR techniques. Our findings reveal that an increase in beam size results in significantly higher VRAM usage. Consequently, increasing the beam size causes numerous out-of-memory crashes and subsequently unduly degrades the performance of LLM-based APR. To address this challenge, we introduced FLAMES —the first APR approach that integrates LLMs with a semantic-guided best-first algorithm to guide the repair process. Our evaluation on various datasets demonstrates that FLAMES substantially outperforms the state-of-the-art baselines. Additionally, FLAMES significantly reduces memory consumption by up to 83% and accelerates the repair process compared to conventional LLM-based APR techniques.

Data Availability. We published replication package including dataset and results and appendix at [3].

REFERENCES

[1] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[2] Toufique Ahmed and Premkumar Devanbu. 2023. Majority Rule: better patching via Self-Consistency. *arXiv preprint arXiv:2306.00108* (2023).

[3] Anonymous Authors. 2023. Replication Package for "Beyond Beam Search: Efficient Large Language Models for Program Repair with Semantic-Guided Patch Generation". https://figshare.com/s/72900c704ce5febfe685.

[4] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[5] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134* (2024).

[6] Lingchao Chen, Yicheng Ouyang, and Lingming Zhang. 2021. Fast and precise on-the-fly patch validation for all. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1123–1134.

[7] Liushan Chen, Yu Pei, Minxue Pan, Tian Zhang, Qixin Wang, and Carlo A Furia. 2022. Program repair with repeated learning. *IEEE Transactions on Software Engineering* 49, 2 (2022), 831–848.

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[9] Zimin Chen, Steve James 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, 09 (2021), 1943–1959.

[10] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2024. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems* 36 (2024).

[11] Saibo Geng & HuggingFace's developers. 2024. Sequential beam search(a.k.a Low-memory beam search). https://github.com/huggingface/transformers/pull/26304.

[12] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 302–313.

[13] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 422–431.

[14] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533* (2023).

[15] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[16] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027* (2020).

[17] Xiang Gao, Yannic Noller, and Abhik Roychoudhury. 2022. Program repair. *arXiv preprint arXiv:2211.12787* (2022).

[18] 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*. 19–30.

[19] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[20] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An empirical study on fine-tuning large language models of code for automated program repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1162–1174.

[21] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-Coder Technical Report. *arXiv preprint arXiv:2409.12186* (2024).

[22] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1430–1442.

[23] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1251–1263.

[24] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

[25] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1646–1656.

[26] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA)*. 437–440.

[27] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* 4 (1996), 237–285.

[28] Sungmin Kang and Shin Yoo. 2022. Language models can prioritize patches for practical program patching. In *Proceedings of the Third International Workshop on Automated Program Repair*. 8–15.

[29] Milod Kazerounian, Jeffrey S Foster, and Bonan Min. 2021. SimTyper: sound type inference for Ruby using type equality prediction. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.

[30] YoungJae Kim, Seungheon Han, Askar Yeltayuly Khamit, and Jooyong Yi. 2023. Automated program repair from fuzzing perspective. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 854–866.

[31] YoungJae Kim, Yechan Park, Seungheon Han, and Jooyong Yi. 2024. Enhancing the Efficiency of Automated Program Repair via Greybox Analysis. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE 2024)*.

[32] Serkan Kirbas, Etienne Windels, Olayori McBello, Kevin Kells, Matthew Pagano, Rafal Szalanski, Vesna Nowack, Emily Rowan Winter, Steve Counsell, David Bowes, et al. 2021. On the introduction of automatic program repair in Bloomberg. *IEEE Software* 38, 4 (2021), 43–51.

[33] Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*. Springer, 282–293.

[34] Xuan-Bach D Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. 2019. On reliability of patch correctness assessment. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 524–535.

[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 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. Association for Computing Machinery, 593–604.

[36] Xuan-Bach D Le and Quang Loc Le. 2021. Refixar: Multi-version reasoning for automated repair of regression errors. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 162–172.

[37] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 213–224.

[38] Xuan-Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*. 163–163.

[39] Thanh Le-Cong, Hong Jin Kang, Truong Giang Nguyen, Stefanus Agus Haryono, David Lo, Xuan-Bach D Le, and Quyet Thang Huynh. 2022. Autopruner: transformer-based call graph pruning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 520–532.

[40] Thanh Le-Cong, Xuan Bach D Le, Quyet Thang Huynh, and Phi Le Nguyen. 2021. Usability and aesthetics: Better together for automated repair of web pages. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 173–183.

[41] Thanh Le-Cong, Thanh-Dat Nguyen, Bach Le, and Toby Murray. 2025. Towards Reliable Evaluation of Neural Program Repair with Natural Robustness Testing. *ACM Trans. Softw. Eng. Methodol.* (Feb. 2025). https://doi.org/10.1145/3716167 Just Accepted.

[42] 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. *Advances in Neural Information Processing Systems* 36 (2023), 21558–21572.

[43] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th International Symposium on Software Testing and Analysis (ISSTA)*. Association for Computing Machinery, 31–42.

[44] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. Association for Computing Machinery, 615–627.

[45] 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 (ISSTA)*. IEEE, 101–114.
[46] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. Sapfix: Automated end-to-end repair at scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 269–278.
[47] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, and Chunming Hu. 2023. Template-based neural program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1456–1468.
[48] Martin Monperrus, Matias Martinez, He Ye, Fernanda Madeiral, Thomas Durieux, and Zhongxing Yu. 2021. Megadiff: A dataset of 600k java source code changes categorized by diff size. *arXiv preprint arXiv:2108.04631* (2021).
[49] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[50] Huy Nguyen, Christoph Treude, and Patanamon Thongtanunam. 2024. Encoding Version History Context for Better Code Representation. *arXiv preprint arXiv:2402.03773* (2024).
[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 preprint arXiv:2203.13474* (2022).
[52] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust enhancement issues in program repair. In *Proceedings of the 44th International Conference on Software Engineering*. 2228–2240.
[53] Rishov Paul, Md Mohib Hossain, Mohammed Latif Siddiq, Masum Hasan, Anindya Iqbal, and Joanna CS Santos. 2023. Enhancing Automated Program Repair through Fine-tuning and Prompt Engineering. *arXiv preprint arXiv:2304.07840* (2023).
[54] Yuhua Qi, Xiaoguang Mao, and Yan Lei. 2013. Efficient automated program repair through fault-recorded testing prioritization. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 180–189.
[55] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. Association for Computing Machinery, 24–36.
[56] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
[57] André Silva, Sen Fang, and Martin Monperrus. 2023. RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair. *arXiv preprint arXiv:2312.15698* (2023).
[58] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815* (2017).
[59] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).
[60] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.
[61] 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.
[62] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.
[63] 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*. 172–184.
[64] 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*. 1–11.
[65] Marco A Wiering and Martijn Van Otterlo. 2012. Reinforcement learning. *Adaptation, learning, and optimization* 12, 3 (2012), 729.
[66] Emily Winter, David Bowes, Steve Counsell, Tracy Hall, Sæmundur Haraldsson, Vesna Nowack, and John Woodward. 2023. How do developers really feel about bug fixing? Directions for automatic program repair. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1823–1841.
[67] Emily Rowan Winter, Vesna Nowack, David Bowes, Steve Counsell, Tracy Hall, Sæmundur Haraldsson, John Woodward, Serkan Kirbas, Etienne Windels, Olayori McBello, et al. 2022. Towards developer-centered automatic program repair: Findings from bloomberg. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1578–1588.
[68] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. Revisiting the plastic surgery hypothesis via large language models. *arXiv preprint arXiv:2303.10494* (2023).
[69] 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 IEEE/ACM International Conference on Software Engineering (ICSE)* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE, 1482–1494.
[70] 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 (ICSE 2023). Association for Computing Machinery*.
[71] 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.
[72] 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 (ESEC/FSE)*. Association for Computing Machinery, 959–971.
[73] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for 0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).
[74] Yuan-An Xiao, Chenyang Yang, Bo Wang, and Yingfei Xiong. 2023. ExpressAPR: Efficient patch validation for java automated program repair systems. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2038–2041.
[75] Yuan-An Xiao, Chenyang Yang, Bo Wang, and Yingfei Xiong. 2024. Accelerating patch validation for program repair with interception-based execution scheduling. *IEEE Transactions on Software Engineering* (2024).
[76] Qi Xin and Steven P Reiss. 2017. Leveraging syntax-related code for automated program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 660–670.
[77] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 92:1–92:13.
[78] 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 (ICSE)*. IEEE, 1506–1518.
[79] He Ye and Martin Monperrus. 2024. ITER: Iterative Neural Repair for Multi-Location Patches. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.
[80] Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. 2024. ThinkRepair: Self-Directed Automated Program Repair. *arXiv preprint arXiv:2407.20898* (2024).
[81] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2022. Coditt5: Pretraining for source code and natural language editing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
[82] Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. Gamma: Revisiting template-based automated program repair via mask prediction. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 535–547.
[83] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. 2023. Planning with Large Language Models for Code Generation. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=Lr8cOOtYbfL
[84] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. *arXiv preprint arXiv:2404.05427* (2024).
[85] Wenkang Zhong, Chuanyi Li, Kui Liu, Jidong Ge, Bin Luo, Tegawendé F Bissyandé, and Vincent Ng. 2024. Benchmarking and Categorizing the Performance of Neural Program Repair Systems for Java. *ACM Transactions on Software Engineering and Methodology* (2024).
[86] 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 *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Association for Computing Machinery, 341–353.
[87] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877* (2024).
[88] Armin Zirak and Hadi Hemmati. 2024. Improving automated program repair with domain adaptation. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–43.