Unofficial template. This layout is provided for convenience and is **not** affiliated with, endorsed by, or sponsored by ICSE or its organisers. Always check the official call for papers and use the current year's official style files for a real submission.

Towards Automated Program Repair with Large Language Models

Author One[1] Author Two[2] Author Three[1,3]

[1]Department of Computer Science, University of Example, USA [2]School of Software Engineering, Example Institute, Canada [3]Nimbus Research, Redmond, WA, USA

author.one@example.edu

ABSTRACT Automated program repair (APR) aims to reduce the manual effort required to fix software defects. Recent advances in large language models (LLMs) have opened new possibilities for generating candidate patches directly from buggy source code and natural-language descriptions. We present REPAIRLLM, a novel framework that combines static program analysis with in-context learning over repository-scale codebases to generate and validate patches for real-world Java projects. Evaluated on the Defects4J benchmark, REPAIRLLM produces plausible patches for 47 out of 83 bugs in our test set, outperforming the state-of-the-art by 12%. We further introduce a dual-validation pipeline that filters out overfitting patches using spectrum-based fault localisation and mutation testing, raising the rate of correct patches from 58% to 74%.

CCS Concepts: Software and its engineering → Software testing and debugging; Computing methodologies → Natural language generation

1 Introduction

Software maintenance accounts for over 70% of the total lifecycle cost of modern software systems [1]. Among maintenance activities, bug fixing is particularly time-consuming, often requiring developers to understand unfamiliar code written years earlier by different authors. Automated program repair (APR) [6] has emerged as a promising approach to alleviate this burden by automatically generating patches for software defects.

Recent breakthroughs in large language models—from CodeBERT [5] to GPT-style architectures [2]—have demonstrated remarkable capabilities in code generation and understanding tasks. When combined with retrieval-augmented generation and program analysis, these models can produce contextually appropriate patches that respect project-specific coding conventions.

Contributions. This paper makes the following contributions:

1. REPAIRLLM, a novel APR framework that integrates program slicing, fault localisation, and LLM-based patch generation.
2. A dual-validation pipeline that uses both differential testing and mutation analysis to filter out overfitting patches.
3. An empirical evaluation on 83 real-world Java bugs from Defects4J, demonstrating a 12% improvement in plausible-patch rate over the best prior system.

2 Background and Related Work

2.1 Automated Program Repair

APR techniques broadly fall into three categories. *Search-based* approaches [8] explore a space of program mutations guided by fitness functions informed by test suites. *Semantics-based* methods [10] use symbolic execution or constraint solving to synthesise patches that satisfy extracted correctness conditions. *Learning-based* techniques [3] frame patch generation as a neural machine translation problem, mapping buggy code to fixed code.

2.2 Large Language Models for Code

Transformer-based models pretrained on large corpora of source code have shown strong performance on code completion [4], program translation [11], and test-case generation [13]. Our work builds on this line of research by incorporating program-analysis signals into the LLM prompting pipeline.

3 Approach

REPAIRLLM operates in three stages, illustrated in Figure 1.

Stage 1	Stage 2	Stage 3
Fault Localisation (spectrum-based)	Patch Generation (LLM + retrieval)	Patch Validation (mutation analysis)

Figure 1: High-level overview of the REPAIRLLM pipeline.

3.1 Fault Localisation

Given a test suite where some tests fail, we compute the Ochiai suspiciousness score for each statement in the program:

$$\text{suspiciousness}(s) = \frac{\text{failed}(s)}{\sqrt{\text{total\_failed} \cdot (\text{failed}(s) + \text{passed}(s))}} \tag{1}$$

Statements are ranked by suspiciousness, and the top-k statements (k=10 in our experiments) form the fault-localisation window passed to the patch-generation stage.

3.2 Patch Generation

We construct a prompt containing: (1) the buggy method and its enclosing class, (2) the top-k suspicious statements highlighted with inline comments, (3) passing and failing test cases as few-shot examples, and (4) up to five semantically

similar methods retrieved from the project repository using a code-embedding index.

The LLM generates a set of candidate patches, each represented as a unified diff. We restrict the output to the buggy method to keep the search space tractable.

3.3 Dual Validation

Each candidate patch is first validated against the project's test suite. Patches that pass all tests are then subjected to mutation analysis: we seed artificial faults into the patched code and measure whether the test suite detects them. Patches that survive mutation testing with a score below a threshold (θ=0.85) are rejected as likely overfitting [12].

4 Experimental Setup

4.1 Research Questions

We investigate the following research questions:

RQ1 How does REPAIRLLM compare to state-of-the-art APR tools in terms of plausible-patch rate?

RQ2 Does the dual-validation pipeline effectively filter overfitting patches?

RQ3 What is the contribution of program slicing and repository-scale retrieval to patch quality?

4.2 Dataset and Baselines

We evaluate on Defects4J v2.0, using the standard 83-bug subset from the Closure, Lang, Math, and Time projects. Baselines include SimFix [7], TBar [9], SequenceR [3], and selfAPR [14].

5 Results

5.1 Plausible-Patch Rate (RQ1)

Table 1 reports the number of bugs for which each tool produces at least one plausible patch (passing all tests).

Table 1: Number of bugs with plausible patches (Defects4J, 83 bugs).

Tool	Plausible Patches
SimFix	27
TBar	31
SequenceR	34
selfAPR	42
RepairLLM	**47**

REPAIRLLM produces plausible patches for 47 bugs, a 12% improvement over selfAPR (42) and a 38% improvement over SequenceR (34). The gains are most pronounced on Closure (14 vs. 10 for selfAPR), where repository-scale retrieval provides meaningful context for inner-class and anonymous-class patterns.

5.2 Patch Correctness (RQ2)

Manual inspection (two authors, Cohen's $\kappa = 0.81$) shows that the dual-validation pipeline raises the correct-patch rate from 58% (plausible-only) to 74%. Mutation testing eliminates 17 overfitting patches while incorrectly rejecting only 2 correct patches.

5.3 Ablation Study (RQ3)

Removing program slicing reduces plausible patches from 47 to 39 (-17%). Removing repository-scale retrieval reduces performance to 36 patches (-23%), confirming that both components contribute meaningfully to the system's effectiveness.

6 Discussion

6.1 Threats to Validity

Internal. Our manual patch-inspection process, despite high inter-rater agreement, inherently involves subjective judgments about patch semantics. *External.* Defects4J, while widely used, contains predominantly small-to-medium-sized Java projects; results may not generalise to industrial-scale monorepos. *Construct.* The Ochiai metric is one of many spectrum-based formulas; alternative localisation strategies may yield different rankings.

6.2 Implications for Practice

The 12% improvement in plausible-patch rate suggests that LLM-based APR is approaching the point where it can serve as a practical developer assistant. However, the 74% correct-patch rate is still insufficient for fully automated deployment without human review.

7 Conclusion and Future Work

We presented REPAIRLLM, an automated program repair framework that combines spectrum-based fault localisation, LLM-based patch generation with repository-scale retrieval, and dual validation via mutation testing. On the Defects4J benchmark, REPAIRLLM achieves a plausible-patch rate of 47/83, outperforming prior work by 12%, and raises correct-patch precision to 74%.

Future work includes extending REPAIRLLM to multi-language projects (Python, JavaScript), incorporating dynamic execution feedback into the patch-generation loop, and exploring fine-tuning strategies that specialise the LLM for repair-specific objectives.

Data Availability

Our implementation and experimental data are available at `https://github.com/repairllm/icse2026`.

References

[1] B. W. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.

[2] T. Brown *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 1877–1901, 2020.

[3] Z. Chen, S. Komrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "SequenceR:

Sequence-to-sequence learning for end-to-end program repair," *IEEE Trans. Softw. Eng.*, vol. 47, no. 9, pp. 1943–1959, 2021.

[4] M. Chen *et al.*, "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021.

[5] Z. Feng *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of EMNLP*, pp. 1536–1547, 2020.

[6] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[7] J. Jiang, Y. Xiong, H. Zhang, L. Zhang, and Z. Hu, "Shaping program repair space with existing patches and similar code," in *Proc. ISSTA*, pp. 298–309, 2018.

[8] X.-B. D. Le, D. Lo, and C. Le Goues, "History driven automated program repair," in *Proc. SANER*, pp. 213–224, 2015.

[9] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting template-based automated program repair," in *Proc. ISSTA*, pp. 31–42, 2019.

[10] S. Mechtaev, J. Yi, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *Proc. ICSE*, pp. 129–139, 2018.

[11] B. Rozière, M.-A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 20601–20611, 2020.

[12] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? Overfitting in automated program repair," in *Proc. ESEC/FSE*, pp. 532–543, 2015.

[13] M. Tufano *et al.*, "Unit test case generation with transformers and focal context," arXiv preprint arXiv:2009.05617, 2020.

[14] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "SelfAPR: Self-supervised program repair with test execution diagnostics," in *Proc. ASE*, pp. 92–104, 2021.