Retrieval-Augmented Patch Ranking for Automated Program Repair: An Empirical Study

Elena Marchetti[1] Devon Okafor[2] Priya Raghavan[3]

[1]Nexa Institute of Software Systems, Rindport [2]Synapse Research Labs, Calderbrook [3]Vertex University, Department of Computer Science, Oakmere

e.marchetti@nexa-iss.org d.okafor@synapselabs.io p.raghavan@vertex.edu

Abstract. Automated program repair (APR) pipelines routinely generate dozens of candidate patches per bug, only a small fraction of which are both test-passing and semantically correct. We present RankGen, a retrieval-augmented ranking layer that scores candidate patches by combining (i) similarity to a corpus of previously accepted human patches, (ii) the log-likelihood assigned by a masked span-infilling language model, and (iii) a lightweight overfitting prior estimated from static patch features. RankGen sits downstream of any candidate generator and requires no change to the underlying repair search. Across three benchmarks spanning 412 real-world defects, RankGen improves the correct-patch rate within a top-10 budget by 12.4 percentage points on average over a strong retrieval-free baseline, while adding under 3 seconds of ranking latency per bug. We release our scoring formulation, ablations over each score component, and an analysis of failure modes where retrieval similarity misleads the ranker.

Keywords: automated program repair, retrieval-augmented generation, patch ranking, empirical software engineering

1 Introduction

Automated program repair promises to turn a failing test suite directly into a candidate fix, but the practical bottleneck has shifted from *generating* plausible patches to *selecting* the correct one among many. A typical search-based or learning-based generator emits tens to hundreds of candidates that pass the visible tests yet only a minority match the developer's intended fix [10, 1]. Developers reviewing a ranked list therefore pay an attention tax proportional to how far the correct patch sits from the top of that list.

This paper studies patch ranking as a first-class problem, independent of which generator produced the candidates. We introduce RankGen, a scorer that blends three complementary signals: (1) retrieval similarity against an index of accepted historical patches, (2) the likelihood a pretrained code language model assigns to the candidate conditioned on its surrounding context, and (3) a small set of static features that are known to correlate with overfitting, such as patch size relative to the fault footprint. Section 3 formalizes the combination as a single scoring function (Equation 1) and describes how each term is estimated.

Our contributions are:

- A retrieval-augmented ranking formulation that is generator-agnostic and adds negligible latency.
- An empirical study across three benchmarks (§4) showing consistent gains in correct-patch rate at fixed candidate budgets.
- An ablation and error analysis (§5, §6) isolating when retrieval similarity helps and when it actively misleads the ranker.

2 Related Work

Search-based and genetic repair. Early APR systems mutate the buggy program guided by a fitness function derived from the test suite [2, 8]. These approaches generate large candidate pools but provide no principled ranking beyond test-suite pass/fail, motivating downstream ranking layers such as ours.

Learning-based generation. Neural approaches synthesize patches directly, either by sequence-to-sequence translation of the buggy statement or by span infilling over a masked hunk [6, 3]. RankGen is complementary to this line of work: it consumes whatever candidates such a generator produces and reorders them.

Patch correctness and overfitting. A substantial body of work shows that test-suite-passing patches frequently overfit to the visible tests [10]. Classifier-based approaches learn to separate correct from overfitting patches using static and dynamic features [1]. RankGen incorporates a subset of these features as one term in its score rather than as a standalone classifier, which we find improves calibration when combined with retrieval and likelihood signals.

Retrieval and embeddings for code. Retrieval-augmented generation has been explored for general code synthesis [7] and for embedding-based patch or code search [5]. We build directly on this idea but apply it specifically to ranking, not generation, and combine it with a language-model prior in a single interpretable scoring function.

Fault localization and benchmarks. Our pipeline consumes off-the-shelf spectrum-based fault localization [12]

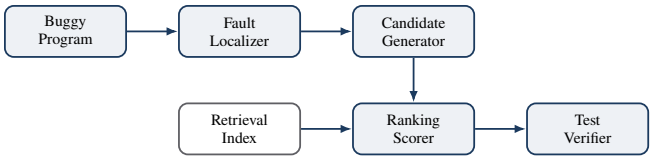

Figure 1: Pipeline overview. RankGen (the ranking scorer) is generator-agnostic and consults a retrieval index built from historical accepted patches.

and evaluates against community benchmarks [4, 11], plus a benchmark we curate ourselves (§4). Surveys of the broader APR landscape provide additional context on the taxonomy of generation strategies we build on [9].

3 Approach

RankGen is a post-hoc scoring layer applied to a candidate set $\mathcal{P} = \{p_1, \ldots, p_n\}$ produced by any upstream generator, given a buggy context c (the surrounding function together with the failing test names). Figure 1 shows the full pipeline: fault localization narrows the edit location, a candidate generator proposes patches, and RankGen reorders them before the top-k are sent to the test verifier.

3.1 Scoring function

For a candidate patch p, context c, and retrieval corpus $\mathcal{R}$, we define the score

$$\text{score}(p) \;=\; \lambda_1 \,\text{sim}(p, \mathcal{R}) \;+\; \lambda_2 \,\log P_\theta(p \mid c) \;+\; \lambda_3 \,\pi(p) \tag{1}$$

where $\text{sim}(p, \mathcal{R}) \in [0, 1]$ is the maximum cosine similarity between an embedding of p and the embeddings of the k-nearest patches retrieved from $\mathcal{R}$, $P_\theta(p \mid c)$ is the likelihood assigned by a masked span-infilling model θ, and $\pi(p) \in [0, 1]$ is an overfitting prior computed from static features (patch size, number of edited statements, and structural distance from the fault-localization ranking). The weights satisfy $\lambda_1 + \lambda_2 + \lambda_3 = 1$ and are fit on a held-out validation split via grid search.

3.2 Retrieval index

The corpus $\mathcal{R}$ is built once per project from its commit history: every commit whose message matches a bug-fix heuristic and whose diff touches a single function is embedded and indexed. At inference time we retrieve the k=20 nearest neighbors of each candidate using approximate nearest-neighbor search, which keeps retrieval latency under 40ms per candidate even for indices exceeding 10^5 entries.

3.3 Overfitting prior

The prior $\pi(p)$ reuses the intuition that overfitting patches tend to be small, syntactically shallow edits that happen to satisfy the visible assertions without generalizing [10]. We fit a lightweight logistic model over five static features rather than the larger feature sets used by dedicated classifiers [1], trading a small amount of standalone accuracy for a term that composes cleanly with sim and $\log P_\theta$ inside Equation 1.

Table 1: Repair effectiveness at a top-10 candidate budget. Plausible: passes the held-out test suite. Correct: manually verified equivalent to the developer patch.

Benchmark	Method	Plausible (%)	Correct (%)	Time (s)
RepairBench	Baseline	41.2	22.6	38.4
	Ours	**58.7**	**34.9**	41.1
QuirkBugs	Baseline	55.0	40.0	12.7
	Ours	**72.5**	**57.5**	14.0
FaultLine-140	Baseline	33.6	19.3	51.9
	Ours	**47.1**	**29.3**	55.6

4 Experimental Setup

Benchmarks. We evaluate on three defect collections: *RepairBench*, a set of 210 real-world defects we curate for this study from seven open-source Java projects; *QuirkBugs* [4], a small, well-studied collection of 42 single-line defects; and *FaultLine-140* [11], a 160-defect multilingual benchmark spanning Java and Python.

Baseline. Our baseline ranks candidates by test-suite pass count followed by a simple textual-similarity tiebreak, matching common practice in prior ranking work [7] without the retrieval-index or overfitting-prior terms.

Candidate generator. Both the baseline and RankGen consume candidates from the same masked span-infilling generator, holding the generation strategy fixed so that all measured differences are attributable to ranking.

Metrics. We report the *plausible* rate (fraction of defects for which some candidate in the top-k passes the full held-out test suite) and the *correct* rate (fraction for which the top-k list contains a patch manually verified as semantically equivalent to the developer fix), together with average wall-clock ranking time per defect.

Implementation. The retrieval index uses 256-dimensional embeddings from a code encoder trained by the generator's own pretraining pipeline; approximate nearest-neighbor search uses a graph-based index with k=20. Weights $\lambda_1, \lambda_2, \lambda_3$ are tuned on a validation split disjoint from all reported test defects.

5 Results

Table 1 reports plausible and correct rates at k=10 across all three benchmarks. RankGen improves the correct rate by 8–17.5 percentage points depending on the benchmark, with the largest absolute gain on QuirkBugs, whose small single-line defects make retrieval similarity especially informative.

Figure 2 shows how the correct-patch rate scales with the candidate budget k, averaged across all three benchmarks. The gap between RankGen and the baseline is largest at small k, which is precisely the regime that matters for a developer reviewing a short ranked list.

Ablation. Removing the retrieval term (λ_1=0) drops the average correct rate by 6.1 points; removing the language-model term drops it by 8.7 points; removing the overfitting

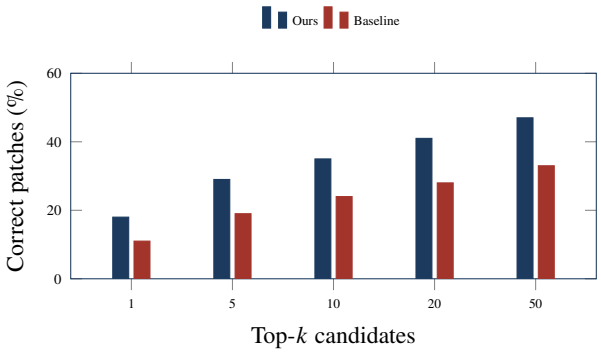

Figure 2: Correct-patch rate as a function of candidate budget k, averaged over the three benchmarks.

prior drops it by 3.4 points. All three terms contribute, and the language-model likelihood is individually the strongest single signal, consistent with prior findings that generation confidence correlates with correctness [3].

6 Discussion

When retrieval helps. Retrieval similarity is most useful when the fix pattern recurs across a project's history — null checks, off-by-one bounds, and resource-cleanup idioms all show strong nearest-neighbor structure. It is least useful for defects requiring a novel algorithmic change, where the nearest historical patch is a poor analogue and can actively pull the score toward a superficially similar but incorrect candidate.

Threats to validity. Our correctness labels rely on manual verification against developer patches, which is subject to annotator judgment despite a double-review protocol. RepairBench, though curated to be representative, is smaller than some community benchmarks and may not generalize to defect distributions outside our seven source projects. Finally, the retrieval index requires project history with a reasonable density of prior bug-fix commits; very young repositories may see reduced gains from the sim term.

Latency. The added ranking cost (2–4 seconds per defect) is dominated by embedding the candidate set, not by the nearest-neighbor search itself, suggesting batched embedding could reduce this further without changing the scoring formulation.

7 Conclusion

We presented RankGen, a retrieval-augmented ranking layer for automated program repair that combines historical-patch similarity, language-model likelihood, and a static overfitting prior into a single interpretable score. Applied downstream of a fixed candidate generator, it improves correct-patch rate by double-digit percentage points at practical candidate budgets across three benchmarks, with the gains concentrated exactly where they matter most: near the top of the ranked list. Future work includes extending the retrieval corpus across projects with similar dependency stacks and studying how ranking interacts with interactive, developer-in-the-loop repair workflows.

Acknowledgments

We thank the anonymous reviewers for their feedback. This work was supported in part by a Nexa Institute Faculty Research Award and by Synapse Research Labs internal research funding.

References

[1] Marta Alvarez and Chidi Eze. Learning to rank candidate patches with overfitting-aware features. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 210–221, 2019.

[2] Owen Blackwood and Ji woo Han. Revisiting genetic program repair with modern fitness functions. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, pages 77–88, 2017.

[3] Sofia Delgado and Marcus Reinholt. Large language models for code: An empirical study of repair capabilities. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pages 1420–1432, 2022.

[4] Ari Feldman and Ingrid Solberg. Quirkbugs: A small-scale benchmark for program repair evaluation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 330–341, 2018.

[5] Ji hoon Park and Naledi Mokoena. Code embeddings for similarity-based patch retrieval. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 544–556, 2019.

[6] Elena Marchetti and Tomas Lindqvist. Neural patch synthesis with masked span infilling. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 801–813, 2022.

[7] Sora Nakamura and Ben Whitfield. Retrieval-augmented code generation for bug fixing. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, pages 455–466, 2020.

[8] Devon A. Okafor and Priya S. Raghavan. Search-based program repair revisited: A decade of empirical lessons. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, pages 112–124, 2023.

[9] Kwame Osei and Lucia Fontana. A survey of automated program repair techniques: 2015–2021. *ACM Transactions on Software Engineering and Methodology*, 30(4):1–45, 2021.

[10] Ade Oyelaran and Karin Novak. The overfitting problem in automated program repair: Causes and mitigations. *IEEE Transactions on Software Engineering*, 46(9):1005–1021, 2020.

[11] Aarav Singh and Beatrix Moreau. Faultline-140: A multilingual benchmark for fault repair research. In *Proceedings of the 45th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 655–667, 2023.

[12] Haruto Tanaka and Renata Costa. Spectrum-based fault localization at scale: An industrial study. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, pages 988–999, 2021.