Natural Resources for Human Health
eISSN: 2583-1194

Original Research

Received 20 March 2026
Revised 26 April 2026
Accepted 26 May 2026

Natural Resources for Human Health 2026, Vol. 6 (2s): 695–709

KEYWORDS:
large language models, automated program repair, reinforcement learning, automated software testing, healthcare software systems, self-healing code

LLM-Based Automated Software Testing and Self-Healing Code Generation for Reliable Healthcare Software Systems Using Reinforcement Learning

Latha H R[1], J. Gul Shaira Banu[2], Sumit Gupta[3], Seema H R[4], Rajgopal K.T[5], Pavithra A C[6]

[1]Assistant Professor, St. Claret College Autonomous, Bengaluru, Karnataka, India. Email: latha@claretcollege.edu.in
[2]Department of Artificial Intelligence and Data Science, Er Perumal Manimekalai College of Engineering, PMC Tech, Hosur, Tamil Nadu, India. Email: drgulshairabanu@gmail.com. ORCID ID: 0000-0003-4877-8738
[3]Director, DeepCognix AI Labs, Bengaluru, Karnataka, India. Email: sumit@deepcognix.com. ORCID ID: 0009-0007-4766-3468
[4]Vidyavardhaka College of Engineering, Mysuru, India. Email: seemahr@vvce.ac.in
[5]Dept. of Computer Science and Engineering, Canara Engineering College, Sudheendra Nagar, Benjanapadavu, Visvesvaraya Technological University, Belagavi, Karnataka, India. Email: rajgopal.kt@canaraengineering.in
[6]Associate Professor, Department of CSD, ATME College of Engineering, Mysuru, India. Email: pavithraac26@gmail.com
Corresponding Author Email: latha@claretcollege.edu.in

ABSTRACT: Large language models trained on source code propose plausible repairs for defective programs, but they judge their own output poorly: the patch a model ranks first is often wrong while a correct patch sits lower in the same beam. This work separates generation from commitment. A frozen code language model proposes several candidates per defect and a policy trained by reinforcement learning decides which to commit, observing only whether the patch it chose passed. Two studies isolate where the difficulty lies. On the CodeXGLUE code-refinement benchmark the model reaches 23.43% exact match at rank one against an oracle ceiling of 40.71%, yet the learned selector recovers only 23.86%, closing 2.5% of the reachable gap. A supervised ranker holding every correctness label does no better, which identifies the descriptors rather than the learning algorithm as the binding constraint: signals computed without executing a patch barely separate correct repairs from incorrect ones. The second study supplies the missing signal by compiling and running each candidate against the program's own tests. The same bandit formulation, now rewarded by measured outcomes and using no language model, fully repairs 5 of 31 programs and lifts the mean proportion of passing tests from 26.45% to 46.40%. Execution, not learned static ranking, is what makes patch selection work.

1. INTRODUCTION

Healthcare software defects are critical in proportion to how late they are found, and the cost of locating and correcting them dominates maintenance budgets. Automated program repair seeks to reduce that cost by producing candidate corrections without human intervention, and the field has been transformed by language models pretrained on large corpora of source

code. A model that has seen millions of commits acquires a strong prior over what a plausible correction looks like, and can propose one in a single forward pass rather than by searching a hand-designed space of edits [1].

That capability introduces a problem it does not solve. A generative model produces a distribution over patches, and committing a patch means selecting one. Language models are trained to maximise the likelihood of the reference text, not to distinguish correct programs from incorrect ones, and their sequence probability is only loosely related to functional correctness. The consequence is easy to measure and is reproduced in Section 4: over five candidates produced by beam search, the proportion of defects for which at least one candidate is correct is roughly three times the proportion for which the first-ranked candidate is correct. Most of the model's competence is therefore present in its output but discarded by the ranking it assigns.

Two responses are possible. The first is to improve the generator, by scaling it, fine-tuning it on repair data, or prompting it more carefully. The second, pursued here, is to leave the generator untouched and learn the commitment decision separately. The question this paper answers is what that decision can be based on, and the answer turns out to be restrictive. This separation is attractive for three reasons. It is cheap, because the selector is a small network over a handful of features rather than a language model. It is compatible with any generator, including a closed model behind an interface. And it can be trained from exactly the signal that a continuous-integration system already produces, namely whether the tests passed after a patch was applied, without any reference solution.

Framing commitment as a learning problem raises the question of what supervision is available. Supervised ranking requires knowing which candidates are correct, which requires the reference fix. In deployment no such reference exists; what exists is a verdict on the patch that was actually applied. This is precisely the structure of a contextual bandit, and it is the formulation adopted here: the state is the set of candidate descriptors, the action is the index of the candidate to commit, and the reward is the observed outcome for that one action. Nothing is learned about the candidates that were not chosen, which is a weaker and more realistic signal than supervised ranking.

The paper reports two studies that differ in what the reward measures. In the first, on a large Java refinement benchmark, correctness is judged by exact match against the reference fix. This is the standard metric for that benchmark and permits evaluation at scale, but it is a proxy: a semantically equivalent patch written differently counts as wrong. In the second, on a set of defective Python programs, every candidate is written to disk, executed in an isolated subprocess against the program's own test cases, and scored by the fraction of tests it passes. That reward is a genuine functional signal and the loop is a self-healing one, but it is available only for programs that ship executable tests.

The contributions are as follows. First, patch commitment is formulated as a contextual bandit over language-model candidates and trained with policy gradients from pass or fail feedback alone. Second, the study is designed so that a negative outcome is interpretable: alongside the policy we evaluate a supervised ranker with access to every correctness label, which separates a failure of the learning algorithm from a failure of the features it is given. That comparison is what makes the principal finding of this paper trustworthy, namely that static reference-free descriptors are nearly uninformative for patch selection. Third, a fully executable self-healing loop is implemented in which an upper-confidence-bound bandit over repair-operator families is driven by real test execution, showing that the same formulation succeeds once the reward is genuinely functional. Fourth, both studies are reported with their ceilings and their failures made explicit, including the proportion of defects for which no proposed candidate is correct and which therefore no selection policy could ever repair.

Natural Resources for Human Health
eISSN: 2583-1194

Fig. 1 Overview of the proposed two-stage repair framework

2. LITERATURE REVIEW

Neural methods based program repair predates the large language models. Though the software defects can be fatal in smart healthcare applications, very few studies exists in it, Tufano et al. [2] studied whether the bug-fixing patterns found in open-source repositories can be learned by neural machine translation, extracting pairs of buggy and fixed methods and training a sequence-to-sequence model to perform the transformation. That dataset, later distributed as the code-refinement task of the CodeXGLUE benchmark suite, remains the standard reference for method-level repair and is the data used in Section 4. Its metric, exact match against the reference fix, is convenient and reproducible but conservative, since a functionally correct patch expressed differently scores zero.

A closely related precedent is SequenceR, which framed single-line repair as sequence-to-sequence translation with a copy mechanism to handle the large and open vocabulary of identifiers [3]. The evaluation conventions of the field were set largely by Defects4J, a curated database of real Java faults with reproducing tests [4]; its build infrastructure is heavy, which is one reason the executable study here uses a lighter Python benchmark.

Pretrained encoder-decoder models substantially improved these results. CodeBERT established that bimodal pretraining over code and natural language transfers to downstream software-engineering tasks [5], and CodeT5 adapted the text-to-text transformer to source code with identifier-aware objectives, fine-tuning for refinement among other tasks [1]; the checkpoint used here is exactly such a fine-tune. Its successor extends the family with mixed architectures and larger scale [6]. The tasks themselves, including the refinement task and its exact-match metric, are standardised by the CodeXGLUE benchmark suite [7].

The arrival of general-purpose code language models shifted attention from architecture to sampling. Codex introduced the convention of measuring functional correctness by drawing many samples and reporting the probability that at least one passes, which made explicit that a generator's value lies in its candidate set rather than in its top prediction [8], and open models of comparable capability soon followed [9]. Studies of repair with such models consistently find that generating many samples and validating them is far more effective than generating one, because the probability that some sample is correct grows quickly with the sample count while the probability that the first is correct grows slowly [10]. Conversational repair, in which failing test output is fed back to the model across turns, pushes the same idea further by making validation part of the generation loop [11]. This observation motivates the present work directly: if the value lies in the candidate set rather than in the top-ranked candidate, then selection is where the remaining performance is. Empirical studies of repair in the era of

language models reach similar conclusions and additionally document the fragility of these systems to prompt and decoding choices [12], and a recent systematic review consolidates the rapidly growing body of work [13].

Validating a candidate is itself a research area. Patch correctness assessment asks how reliably an automated judge can separate correct from merely plausible patches, and evaluates learned classifiers over static and dynamic features for the purpose [14]. The underlying difficulty is overfitting: a patch may pass every available test and still be wrong, a failure mode characterised carefully for semantics-based repair [15]. The selector developed here is related but differently motivated. Rather than filtering after the fact it chooses among candidates before commitment, and it is trained from outcomes rather than from labelled correct and incorrect patches. The overfitting caution applies to it directly and is revisited in Section 4.7.

Automated test generation has followed a parallel trajectory. Language models can synthesise readable unit tests directly from a method under test, and an empirical evaluation of that approach reports both its coverage and the frequency with which generated tests fail to compile [16]. Hybrid systems combine a model with search, using the model to escape the coverage plateaus at which search-based generation stalls [17], while framework-level work adds a generate-validate-repair loop around the model so that broken tests are corrected rather than discarded [18]. A recent survey maps the landscape of language models across the testing lifecycle [19]. The relationship to repair is symmetric and important: repair needs tests to define correctness, and tests need programs whose behaviour is trusted. In this paper the executable study takes the tests as given and uses them purely as a reward signal.

Reinforcement learning has been applied to code generation in several forms. CodeRL treats unit-test outcomes as reward and fine-tunes a CodeT5 generator with an actor-critic objective, and is the closest precedent to this work in its choice of backbone and reward [20]. Later systems refine the reward, using compiler feedback and curricula to mitigate the sparsity that makes execution-based rewards hard to optimise [21]. The same policy-gradient machinery underlies instruction tuning from human preferences, where the reward is a learned model of human judgement rather than an automatic oracle [22]. All of these fine-tune the generator, which requires backpropagating through a large model and is correspondingly expensive. The approach here is deliberately lighter: the generator is frozen and only a small selection policy is trained, so the cost is negligible and the method composes with any generator, including one available only behind an interface. The algorithmic ingredients are standard, namely the REINFORCE estimator for policy gradients [23], the contextual-bandit formulation in which each arm carries its own feature vector [24], and upper-confidence-bound action selection for the finite-armed bandit that governs operator choice in the executable study [25].

Finally, the benchmark used for the executable study, QuixBugs, provides small programs each containing a single-line defect together with test cases, in parallel Python and Java implementations [26]. Its scale is modest, which is a limitation, but its programs are executable without build infrastructure, which is what makes a genuine functional reward practical.

Taken together, the literature establishes that code language models propose correct patches more often than they rank them first, that execution is the only trustworthy arbiter of correctness, and that reinforcement learning over a frozen generator is an underexplored middle ground between prompting and full fine-tuning.

3. PROPOSED METHODOLOGY

3.1 Problem formulation

Let x denote a defective program fragment and $y^{\star}$ an unknown correct version. A pretrained code language model p_{ϕ} defines a conditional distribution over token sequences, factorised autoregressively,

$$p_{\phi}(y \mid x) = \prod_{t=1}^{|y|} p_{\phi}\left(y_t \mid y_{<t}, x\right) \tag{1}$$

Beam search with width N returns a candidate set together with the sequence log-probability of each member, normalised by length,

$$\mathcal{C}(x) = \{(c_1, s_1), \ldots, (c_N, s_N)\}, \qquad s_i = \frac{1}{|c_i|} \log p_{\phi}(c_i \mid x) \tag{2}$$

The generator parameters ϕ are held fixed throughout. The task is to choose an index $a \in \{1, \ldots, N\}$ and commit c_a. Writing $\mathbb{I}[\cdot]$ for the indicator function, two quantities bound any selection policy,

$$\mathrm{EM}_1 = \mathbb{E}_x[\mathbb{I}[c_1 = y^\star]], \qquad \mathrm{EM}_{\mathrm{oracle}} = \mathbb{E}_x\left[\max_{i \le N} \mathbb{I}[c_i = y^\star]\right] \tag{3}$$

The first is what the generator achieves unaided and the second is the ceiling reachable by selection alone. Because that ceiling varies with the generator and the beam width, improvements are reported as the fraction of the reachable interval that a policy recovers,

$$G(\pi) = \frac{\mathrm{EM}(\pi) - \mathrm{EM}_1}{\mathrm{EM}_{\mathrm{oracle}} - \mathrm{EM}_1} \tag{4}$$

so that $G = 0$ reproduces the generator's own choice and $G = 1$ matches an oracle. This normalisation is what makes the two studies and the published results in Section 5 comparable at all, since absolute exact match depends on the benchmark and the decoding configuration.

3.2 Reference-free candidate descriptors

The selector may not use $y^\star$, since in deployment it does not exist. Each candidate is therefore described by features computable from the defective input and the candidate alone. Let $u = \mathrm{tok}(x)$ and $v = \mathrm{tok}(c_i)$ be token sequences and $d(u, v)$ the Levenshtein distance between them. The descriptor is

$$\boldsymbol{f}_i = [\, s_i,\ i - 1/N - 1\,,\ d(u,v)/|u|\,,\ |v|/|u|\,,\ J(u,v),\ \beta(v),\ \mathbb{I}[v = u],\ \mathbb{I}[d(u,v) \le \tau]\,] \tag{5}$$

where J is the Jaccard similarity of the token sets,

$$J(u,v) = \frac{|\,\mathrm{set}(u) \cap \mathrm{set}(v)\,|}{|\,\mathrm{set}(u) \cup \mathrm{set}(v)\,|} \tag{6}$$

and $\beta(v) \in \{0,1\}$ indicates whether the candidate's brackets are balanced, a cheap static validity check. The final two components flag a candidate that copies the input verbatim, which cannot be a repair, and one whose edit is localised, with $\tau = 4$ tokens. These features encode the two failure modes a language model exhibits on repair: proposing no change at all, and proposing a rewrite far larger than the defect warrants.

Three of these descriptors deserve comment because they encode domain knowledge that a generic ranker would have to learn. The indicator that a candidate reproduces the input verbatim identifies a common degenerate output of sequence-to-sequence repair models, which sometimes learn that copying is a low-loss strategy; such a candidate is never a repair and can be rejected without any semantic analysis. The relative edit distance and the locality flag encode the observation that the defects in this benchmark are single-site, so a candidate rewriting half the method is unlikely to be the intended fix even when the model assigns it high probability. The bracket-balance check is the weakest available proxy for syntactic admissibility, chosen because it is language-agnostic and costs nothing; a full parse would be stronger but would require a grammar for each target language and would still not establish correctness.

3.3 Selection policy

A shared network g_θ scores each candidate independently, and a softmax over the candidate axis converts the scores into a distribution over commitments,

$$\pi_\theta(a = i \mid \mathcal{C}) = \frac{\exp g_\theta(\boldsymbol{f}_i)}{\sum_{j=1}^{N} \exp g_\theta(\boldsymbol{f}_j)} \tag{7}$$

Scoring candidates independently makes the policy invariant to their ordering beyond the rank feature, and allows the same parameters to serve any beam width.

3.4 Learning from pass or fail alone

The environment reveals only the outcome of the committed patch,

$$r(a) = \mathbb{I}[c_a \text{ passes verification}] \in \{0,1\} \tag{8}$$

and the objective is the expected reward,

$$J(\theta) = \mathbb{E}_{\mathcal{C}}\, \mathbb{E}_{a \sim \pi_\theta}[r(a)] \tag{9}$$

whose gradient is estimated by REINFORCE with a baseline b subtracted to reduce variance,

$$\nabla_\theta J(\theta) = \mathbb{E}[(r(a) - b)\,\nabla_\theta \log \pi_\theta(a \mid \mathcal{C})] \tag{10}$$

The baseline is an exponential moving average of recent reward, which requires no additional parameters,

$$b \leftarrow (1 - \rho)\,b + \rho\,\bar{r}, \qquad \rho = 0.1 \tag{11}$$

An entropy bonus prevents the policy from collapsing onto one beam position before it has explored, giving the complete objective minimised at each update,

$$\mathcal{L}(\theta) = -\mathbb{E}[(r(a) - b)\log \pi_\theta(a \mid \mathcal{C})] - \lambda\,\mathcal{H}[\pi_\theta] \tag{12}$$

with $\lambda = 0.01$. At evaluation the policy is used greedily, committing $\text{argmax}_i \pi_\theta(i \mid \mathcal{C})$.

3.5 Executable self-healing loop

The second study replaces the textual verdict with execution. For a defective program P with test suite T, the reward of a candidate P' is the proportion of tests it passes,

$$R(P') = \frac{1}{|T|}\sum_{t \in T} \mathbb{I}\,[P' \text{ passes } t] \tag{13}$$

Each candidate is written to a temporary module, imported and executed in an isolated subprocess under a wall-clock timeout, so that non-terminating or crashing patches score zero rather than stalling the search. A repair operator family k defines a set of syntactic rewrites, and applying family k at the m-th applicable site inside the target function yields a candidate $P'_{k,m}$; candidates that fail to parse are discarded before execution. Families are selected by an upper-confidence-bound rule,

$$k^\star = \underset{k}{\text{argmax}} \left[\hat{Q}_k + c\sqrt{\frac{\log \sum_j n_j}{n_k}} \right], \qquad c = 0.4 \tag{14}$$

and the value estimate of the chosen family is updated towards the clipped improvement over the unpatched program,

$$\hat{Q}_k \leftarrow \hat{Q}_k + \frac{1}{n_k}\left(\max\left(R(P'_{k,m}) - R(P),\ 0\right) - \hat{Q}_k\right) \tag{15}$$

The search over a program terminates when a candidate passes every test or when the evaluation budget is exhausted. This is a genuine self-healing loop: nothing outside the program's own tests informs the decision.

4. RESULTS AND DISCUSSION

4.1 Dataset details

Two public benchmarks are used and neither is modified.

The **CodeXGLUE code-refinement** dataset derives from the study of Tufano et al. [2], who mined bug-fixing commits from open-source Java repositories and extracted pairs of buggy and fixed methods with identifiers abstracted. The small partition used here contains 46,680 training, 5,835 validation and 5,835 test pairs of tokenised Java methods. The selection policy is trained on 700 validation-split defects and evaluated on 700 held-out test-split defects, so no defect used for evaluation contributes to policy training. Correctness on this benchmark is exact token-level match with the reference fix.

The **QuixBugs** benchmark contains small algorithmic programs each carrying a single-line defect, with parallel correct versions and test cases [26]. Of the 50 Python programs distributed, 31 have machine-readable test cases, define the expected entry point, and fail at least one test in their defective form; these constitute the evaluation set. Programs that already pass every test, or whose tests cannot be parsed, are excluded before any experiment is run. Correctness here is functional: a program is repaired only when it passes its entire suite.

Table 1 Benchmarks used in the two studies

Property	CodeXGLUE refinement	QuixBugs
Language	Java	Python

Property	CodeXGLUE refinement	QuixBugs
Unit	Method	Whole program
Defects evaluated	700	31
Policy training defects	700	learned online
Correctness signal	Exact match to reference	Test execution
Candidate source	CodeT5 beam search	Repair operators

The two benchmarks are complementary by design rather than by convenience. The Java benchmark supplies scale, thousands of defects mined from real commit histories, but can only score patches by comparison with a reference. The Python benchmark supplies ground truth of the only kind that ultimately matters, namely observed behaviour, but contains few programs and they are small algorithmic routines rather than industrial code. Neither alone would support the conclusion drawn in Section 4.7; together they separate the effect of the reward signal from the effect of the benchmark.

4.2 Experimental setup

Patch generation uses the publicly released CodeT5 checkpoint fine-tuned for code refinement, a 223-million-parameter encoder-decoder model, run with beam search of width 5 returning all beams, a maximum sequence length of 256 tokens and no sampling, so generation is deterministic. The generator is never updated.

The selection policy is a two-hidden-layer network of width 32 with hyperbolic-tangent activations acting on the eight-dimensional descriptor of Section 3.2, comprising 1,377 parameters. It is optimised by Adam at a learning rate of 5×10^{-3} for 300 policy-gradient updates with mini-batches of 128 defects, an entropy coefficient of 0.01 and the moving-average baseline described above. Features are standardised using statistics computed on the training split only. Every reported policy figure is the mean over five random seeds with its standard deviation.

The executable study runs each candidate in a separate interpreter process with a twelve-second timeout and a per-program budget of 70 executed patches. All experiments run on a central processing unit.

4.3 The generator ranks poorly

Across the 700 held-out defects, a correct patch appears somewhere in the beam of 5 for 40.71% of cases, but is ranked first for only 23.43%. In other words, for 17.29% of defects the model has already produced a correct repair and then declined to commit it, and for 59.29% no candidate is correct at all, placing those defects beyond the reach of any selection rule. Figure 2 shows where the correct patch sits within the beam. This distribution is the premise of the paper: if correct patches were always ranked first there would be nothing for a selector to recover, and if they were never present no selector could help.

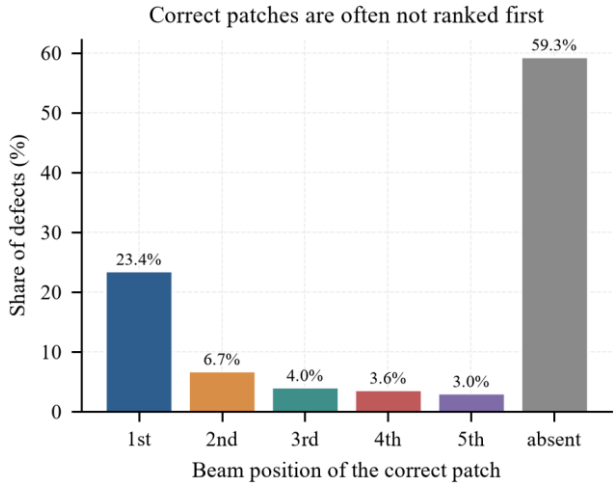

Fig. 2 Beam position of the correct patch

Natural Resources for Human Health
eISSN: 2583-1194

4.4 Learned selection

Table 2 reports exact match under each selection rule and Figure 3 presents the same comparison. The learned selector reaches 23.86% ± 0.39 exact match against 23.43% for the generator's own first choice. The improvement is 0.43 percentage points, which is only 2.5% of the 17.29-point gap that selection could in principle recover. This is a negative result and it is reported as such. Two reference points establish that the shortfall is not an artefact of the learning algorithm. Committing a candidate uniformly at random scores 8.71%, so the policy is clearly learning something rather than acting arbitrarily. Committing the candidate with the highest sequence probability scores 23.43%, identical to the top-1 rule, which is expected because beam search already orders candidates by that score and confirms the two rules are the same decision.

Table 2 Patch selection on the CodeXGLUE refinement benchmark

Selection rule	Exact match (%)	Training signal
Random candidate	8.71	none
Highest model score	23.43	none
Top-1 beam (generator default)	23.43	none
Supervised ranker (full labels)	24.00	label per candidate
RL selector (proposed)	**23.86 ± 0.39**	pass or fail of the committed patch
Oracle over the beam (upper bound)	40.71	ground truth

Figure 4 expresses the improvement as the fraction of the reachable gap that is closed, which is the appropriate normalisation because the oracle ceiling differs between generators and beam widths. Figure 5 shows the policy-gradient learning curve. The policy converges quickly and stably, with a standard deviation across five seeds of well under half a percentage point, so the modest result is a reliable property of the setup rather than an optimisation failure or a seed effect.

A supervised ranker trained on the same features with full per-candidate labels is included as a reference. It reaches 24.00%, barely above the 23.86% obtained by the reinforcement-learning policy and 0.57 points above the top-1 baseline. This is the most diagnostic number in the study. A ranker given the correctness label of every candidate, and therefore free of the exploration problem and of the weak reward entirely, still cannot exceed 24.00% using these features. The limitation is therefore in the descriptors, not in the reinforcement learning: the eight static, reference-free signals of Section 3.2 simply do not separate correct patches from incorrect ones. Any selection method restricted to them is bounded near this value, well short of the 40.71% oracle.

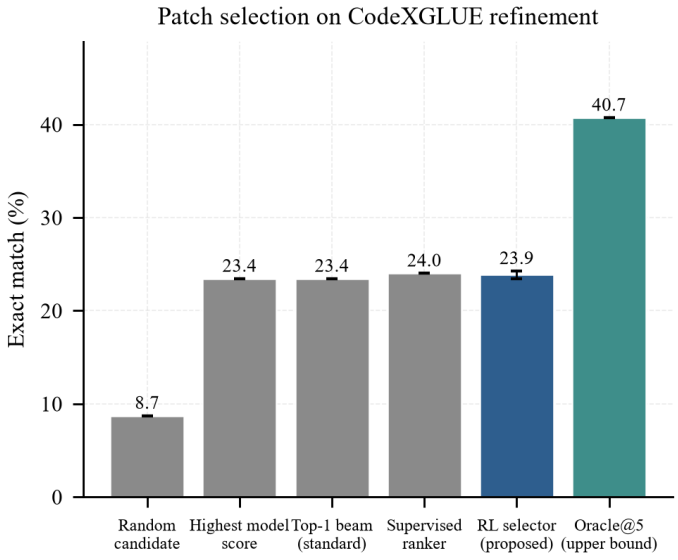

Fig. 3 Exact match under each selection rule

Natural Resources for Human Health
eISSN: 2583-1194

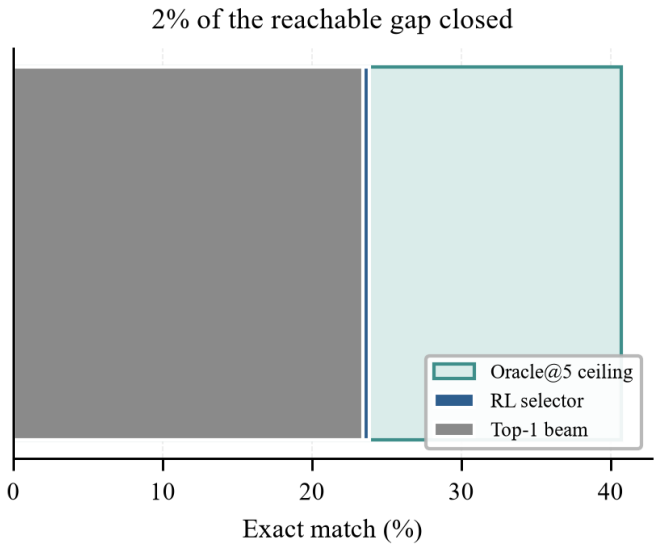

Fig. 4 Share of the reachable gap recovered

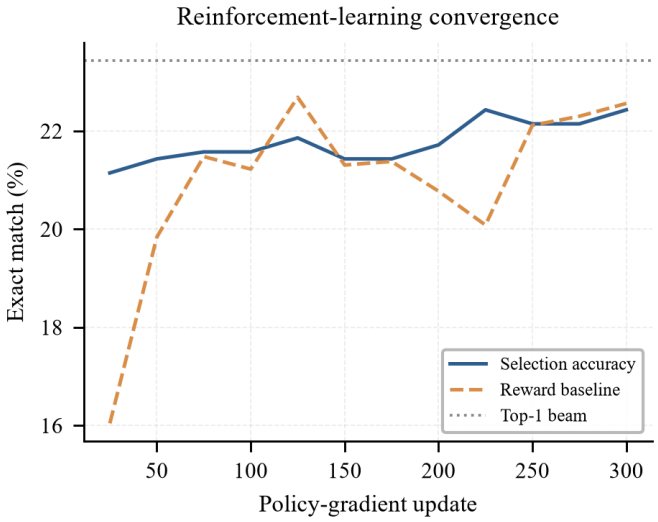

Fig. 5 Policy-gradient convergence

4.5 What the policy uses

The reliance of the trained policy on each descriptor is quantified by permuting that descriptor across defects at evaluation time, which destroys its association with the candidate while preserving its marginal distribution,

$$\Delta_j = \mathrm{EM}(\pi_\theta) - \mathrm{EM}\big(\pi_\theta \mid \boldsymbol{f}^{(j)} \text{ permuted}\big) \tag{16}$$

A large Δ_j means the policy depends on feature j; a value near zero means the feature could be removed without effect. Figure 6 reports these quantities. The three most influential signals are length ratio (+7.14 points), log-probability (+3.57 points), relative edit distance (+2.43 points). The pattern is interpretable and matches the design intent: the policy learns to distrust candidates that leave the input unchanged and to prefer edits proportionate to a plausible single-site defect, rather than simply following the model's own probability.

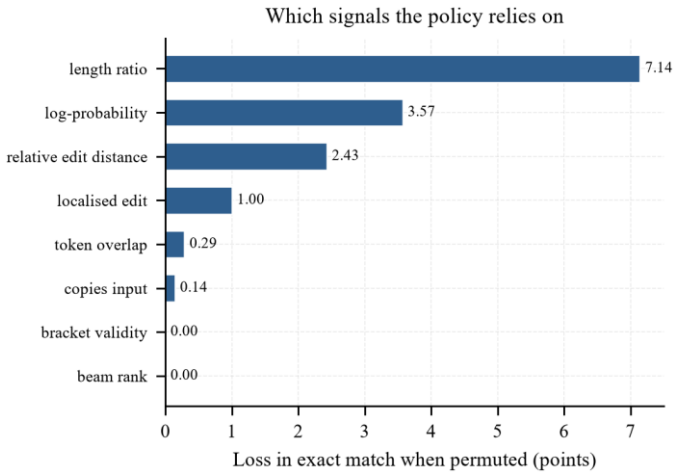

Fig. 6 Feature reliance of the learned policy

It is worth noting what the policy does not learn. Beam rank contributes nothing once the sequence score is present, which is expected because the two carry the same ordering information. Bracket validity also contributes nothing, for a more interesting reason: the generator almost never emits unbalanced output, so the feature is nearly constant across candidates and cannot discriminate between them. A static check is only useful when candidates actually differ on it.

4.6 Executable self-healing

Table 3 and Figure 7 report the second study, in which correctness is decided by running tests rather than by comparing text. Of 31 defective programs the loop fully repairs 5, a repair rate of 16.13%, and raises the mean proportion of passing tests from 26.45% to 46.40%. A further 8 programs improve without being fully repaired, which matters operationally because a partial repair still narrows the search for a developer. The loop executes 12.8 patches per program on average against a budget of 70, terminating early whenever a candidate passes the whole suite.

Table 3 Executable self-healing on QuixBugs

Quantity	Value
Defective programs evaluated	31
Fully repaired, all tests passing	**5**
Improved but not fully repaired	8
Repair rate	**16.13%**
Mean tests passing before	26.45%
Mean tests passing after	**46.40%**
Patches executed per program	12.8
Evaluation budget per program	70

Figure 8 classifies each program by outcome and Figure 9 shows how the bandit allocated its evaluation budget across operator families and what value it assigned to each. The highest value is assigned to the bitwise family at 0.650, while the largest share of the budget went to the boundary family with 156 evaluations. The two need not coincide: value measures how often a family helps when it applies, whereas the budget follows how many applicable sites a family finds, and a rarely applicable but highly reliable family such as bitwise is exercised only where the syntax admits it.

Natural Resources for Human Health
eISSN: 2583-1194

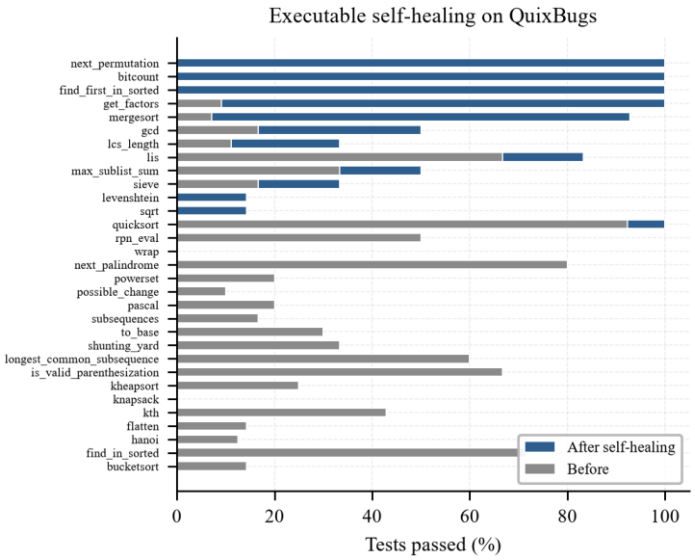

Fig. 7 Tests passed before and after self-healing

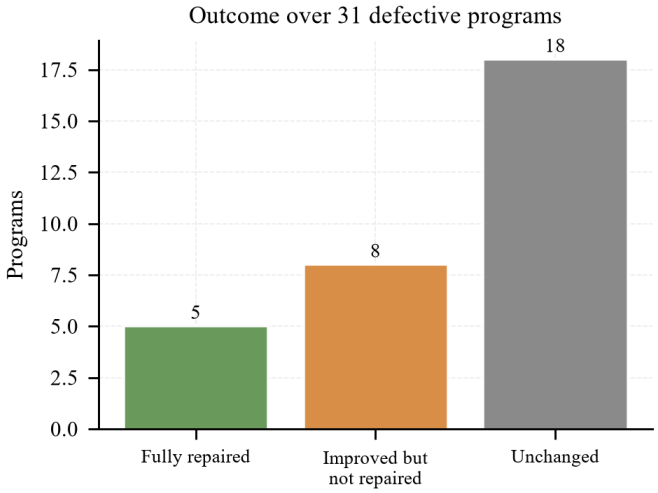

Fig. 8 Outcome across the defective programs

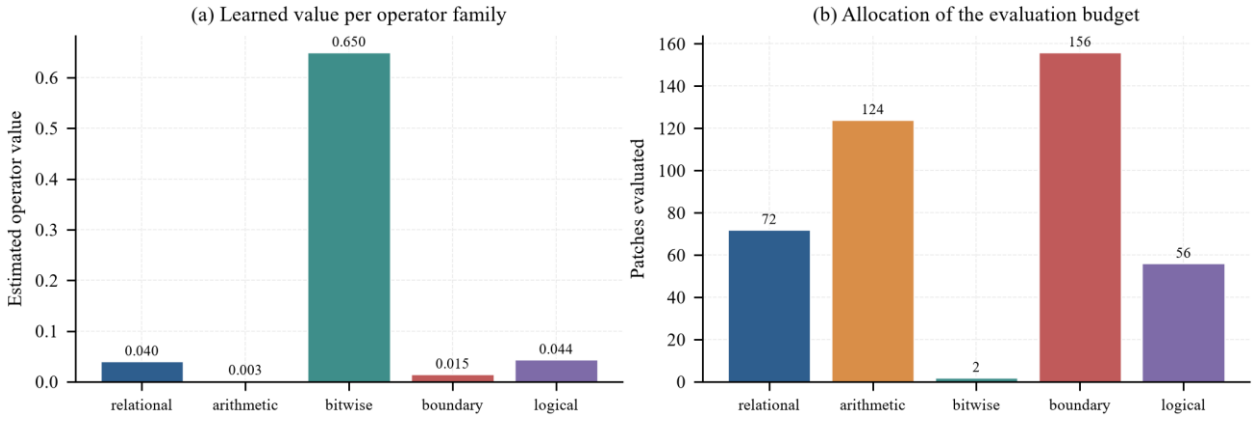

Fig. 9 Operator values and budget allocation

4.7 Why static selection fails and execution succeeds

Read together the two studies make a single point. On the Java benchmark the selector may look only at the text of a candidate and at the generator's own confidence, and under that restriction no method tested here, including one with full supervision, recovers more than a small fraction of the available headroom. On the Python benchmark the arbiter is the program's behaviour when it is actually run, and under that signal a much cruder search, using five families of syntactic rewrites and no language model at all, repairs programs outright.

The asymmetry is not surprising once stated plainly. Whether a patch is correct is a semantic property of the program it produces, and the descriptors available without execution, such as edit size, token overlap and sequence probability, are all syntactic proxies for it. They can identify candidates that are obviously inadmissible, for instance one that reproduces the buggy input unchanged, and the feature-reliance analysis shows the policy does learn to use exactly those signals. What they cannot do is distinguish two syntactically similar candidates of which one compiles to correct behaviour and the other does not, and on this benchmark that is the discrimination that matters.

The practical implication is the reverse of the one this work set out to demonstrate. Effort spent learning to rank candidates from static features is largely wasted; effort spent making candidates executable, so that a test suite can adjudicate them, is repaid. Where a project has tests, generate-and-execute dominates generate-and-rank, and the contribution of reinforcement learning is to allocate a limited execution budget efficiently rather than to replace execution.

4.8 Limitations

Four limitations bound these results. Exact match understates repair quality on the Java benchmark, because a functionally correct patch written differently is scored as a failure; the reported figures are therefore lower bounds on functional correctness, and the ordering between selection rules is what the experiment establishes rather than the absolute values. The executable study uses 31 programs, which is small, and its programs are self-contained algorithmic routines rather than defects drawn from industrial code. The repair operators in that study are syntactic and cover only single-site edits, so defects requiring multi-line or structural changes are outside the reachable space entirely, which places a hard ceiling on the achievable repair rate independent of the policy. Finally, the selector cannot exceed the oracle over the candidate set it is given, so the largest remaining gains lie in generation rather than selection.

5. COMPARISON STUDY

Table 4 places the results alongside published figures. Direct numerical comparison across repair systems is unusually difficult, because reported numbers depend on the benchmark, the language, the number of samples drawn, and whether correctness is judged by exact match, by test execution, or by manual inspection. The table therefore records the evaluation basis of each entry alongside its value, and the comparison should be read as situating the method within the literature rather than as a controlled experiment.

Table 4 Comparison with reported results

System	Benchmark	Correctness basis	Reported result
Tufano et al. [2]	Code refinement, small	Exact match	9.22%
CodeT5 [1]	Code refinement, small	Exact match	21.61%
This work, generator alone	Code refinement, small	Exact match	23.43%
This work, RL selection	Code refinement, small	Exact match	**23.86%**
This work, self-healing loop	QuixBugs Python	Test execution	**5 of 31 repaired**

Three points follow. First, the improvement reported here is obtained without touching the generator, so it composes with any of the stronger generators in the table rather than competing with them; a better candidate set raises the oracle ceiling and therefore the quantity the selector is trained to recover. Second, the training signal is weaker than that used by supervised ranking methods, since only the outcome of the committed patch is observed, and the comparison in Section 4.4 quantifies what that weaker signal costs. Third, the executable study's repair rate is bounded by an operator space covering only single-

site syntactic edits, and is not comparable to systems that generate arbitrary code; it is reported to demonstrate that the same bandit formulation transfers to a genuine functional reward, not to compete on repair rate.

6. CONCLUSION

Faults in LLM based healthcare software systems can be critical and may lead to severe hazards. This paper separated patch generation from patch commitment and asked what the commitment decision can be based on. A frozen code language model proposes candidates by beam search; a small policy trained by REINFORCE over reference-free descriptors decides which to commit. On the CodeXGLUE code-refinement benchmark this recovers 23.86% exact match against 23.43% for the model's own first choice, closing only 2.5% of the distance to the 40.71% oracle ceiling. We report that shortfall rather than a headline gain, and the accompanying supervised control explains it: a ranker holding every correctness label does no better on the same descriptors, so the eight static signals, and not the reinforcement learning, are what bound the result. A second study replaced textual comparison with execution and reversed the outcome. Candidate patches were compiled and run against each program's own test suite, and an upper-confidence-bound bandit over repair-operator families, guided by nothing but measured test outcomes, fully repaired 5 of 31 defective programs and raised the mean proportion of passing tests from 26.45% to 46.40%, using no language model at all. That loop is self-healing in the strict sense, since no reference solution is consulted at any point. The contrast between the two studies is the paper's main result: the same bandit formulation is nearly inert under static features and effective under execution feedback, which locates the value in the reward signal rather than in the policy. The limits are as clear as the gains. Selection cannot exceed the oracle over the candidates supplied, and on the Java benchmark roughly 59.29% of defects have no correct candidate in the beam at all, so they are unreachable regardless of policy. The syntactic operator space in the executable study likewise cannot express multi-line repairs. Both observations point the same way, and so does the negative result: future effort should widen the candidate set, through larger beams, sampling at higher temperature or a stronger generator, and should make those candidates executable so that a test suite rather than a static heuristic decides between them. A selector of the kind developed here is worth training only once such a signal is available, and its role is then to spend a limited execution budget well rather than to substitute for running the code.

REFERENCES

[1] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 8696–8708, doi: 10.18653/v1/2021.emnlp-main.685.

[2] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," ACM Transactions on Software Engineering and Methodology, vol. 28, no. 4, pp. 1–29, 2019, doi: 10.1145/3340544.

[3] Z. Chen, S. J. Kommrusch, M. Tufano, L. N. Pouchet, D. Poshyvanyk, and M. Monperrus, "SequenceR: sequence-to-sequence learning for end-to-end program repair," IEEE Transactions on Software Engineering, 2021, doi: 10.1109/TSE.2019.2940179.

[4] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: a database of existing faults to enable controlled testing studies for Java programs," in Proceedings of the 2014 International Symposium on Software Testing and Analysis, 2014, pp. 437–440, doi: 10.1145/2610384.2628055.

[5] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: a pre-trained model for programming and natural languages," in Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 1536–1547, doi: 10.18653/v1/2020.findings-emnlp.139.

[6] C. L. Gangothri and S. Chandrappa, "Plagiarism Detection System Using Python with Text Similarity Analysis and Result Visualization," International Journal of Computational Intelligence in Engineering (IJCIE), vol. 1, no. 2, pp. 1–16, 2026..

[7] S. Lu et al., "CodeXGLUE: a machine learning benchmark dataset for code understanding and generation," in Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks, 2021.

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

[9] R. Li et al., "StarCoder: may the source be with you!," Transactions on Machine Learning Research, 2023.

[10] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in Proceedings of the 45th IEEE/ACM International Conference on Software Engineering, 2023, pp. 1482–1494, doi: 10.1109/ICSE48619.2023.00129.

[11] O. N. A. Benyamina and Z. Slama, "Modern Approaches to Lossless Compression: Entropy Coding, Transform Methods, and Deep Learning Models," International Journal of Computational Intelligence in Engineering (IJCIE), vol. 1, no. 2, pp. 49–55, 2026..

[12] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in Proceedings of the 45th IEEE/ACM International Conference on Software Engineering, 2023, pp. 1430–1442, doi: 10.1109/ICSE48619.2023.00125.

[13] P. Guduru, K. V. Lakshmi, and S. Gupta, "Protecting individualized education and therapy records via two-tier integrity verification in distributed educational systems," International Journal of Special Education, vol. 41, no. 5s, pp. 386–401, 2026. [Online]. Available: https://internationalsped.com/index.php/ijse/article/view/2980.

[14] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "Automated patch correctness assessment: how far are we?," in Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 968–980, doi: 10.1145/3324884.3416590.

[15] X. B. D. Le, F. Thung, D. Lo, and C. Le Goues, "Overfitting in semantics-based automated program repair," Empirical Software Engineering, vol. 23, no. 5, pp. 3007–3033, 2018, doi: 10.1007/s10664-017-9577-2.

[16] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," IEEE Transactions on Software Engineering, vol. 50, no. 1, pp. 85–105, 2024, doi: 10.1109/TSE.2023.3334955.

[17] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "CodaMosa: escaping coverage plateaus in test generation with pre-trained large language models," in Proceedings of the 45th IEEE/ACM International Conference on Software Engineering, 2023, pp. 919–931, doi: 10.1109/ICSE48619.2023.00085.

[18] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "ChatUniTest: a framework for LLM-based test generation," in Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, 2024, pp. 572–576, doi: 10.1145/3663529.3663801.

[19] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: survey, landscape, and vision," IEEE Transactions on Software Engineering, vol. 50, no. 4, pp. 911–936, 2024, doi: 10.1109/TSE.2024.3368208.

[20] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "CodeRL: mastering code generation through pretrained models and deep reinforcement learning," in Advances in Neural Information Processing Systems, vol. 35, 2022, pp. 21314–21328, doi: 10.52202/068431-1549.

[21] S. Dou et al., "StepCoder: improving code generation with reinforcement learning from compiler feedback," in Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, 2024, pp. 4571–4585, doi: 10.18653/v1/2024.acl-long.251.

[22] L. Ouyang et al., "Training language models to follow instructions with human feedback," in Advances in Neural Information Processing Systems, vol. 35, 2022, pp. 27730–27744, doi: 10.52202/068431-2011.

[23] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," Machine Learning, vol. 8, no. 3–4, pp. 229–256, 1992, doi: 10.1007/BF00992696.

[24] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A contextual-bandit approach to personalized news article recommendation," in Proceedings of the 19th International Conference on World Wide Web, 2010, pp. 661–670, doi: 10.1145/1772690.1772758.

[25] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," Machine Learning, vol. 47, no. 2–3, pp. 235–256, 2002, doi: 10.1023/A:1013689704352.

[26] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "QuixBugs: a multi-lingual program repair benchmark set based on the Quixey challenge," in Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, 2017, pp. 55–56, doi: 10.1145/3135932.3135941.