

Execution-Feedback Code Repair Agents with Test-Guided Iterative Refinement

Brian Thompson¹, Fan Zhang², Melissa Adams³, Jun Zhao⁴

¹Department of Computer Science, University of Southern California, Los Angeles, CA, USA

²Department of Computer Science, Stanford University, Stanford, CA, USA

³Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA

⁴Department of Computer Science, University of California, Los Angeles, CA, USA

brian.thompson88@gmail.com

Abstract

Code agents often produce near-correct programs that fail executable checks. This study evaluates an execution-feedback repair agent (EFR) that converts test outcomes into structured evidence, localizes suspicious code, proposes bounded abstract-syntax-tree edits, and re-ranks remaining patches after failed attempts. All 164 HumanEval tasks were used. After canonical solutions passed the official checks, one reversible fault was introduced per task across six edit families. The 1,582 dynamically executed assertions were split into 1,260 feedback and 322 held-out assertions, and 1,470 localized proposals were executed. A task-grouped five-fold logistic ranker supplied a patch prior; coverage, exception compatibility, and test-score changes supplied runtime evidence. EFR repaired 83.54% of programs within five attempts and 87.80% within ten (95% Wilson interval, 81.91%-91.97%), compared with 82.32% for learned fixed-order search, 66.46% for coverage search, and 57.32% for random search. The ten-attempt policy averaged 2.77 attempts and 914.6 lexical tokens. Only one feedback-passing patch failed held-out verification. The results show that a learned prior improves early efficiency, while bounded execution and evidence updates convert local failures into verified repairs.

Keywords—Code Agents, Automated Program Repair, Execution Feedback, Humaneval, Fault Localization, Iterative Refinement, Test-Guided Repair

INTRODUCTION

Code generation has moved from token completion toward agents that can write a program, invoke an execution environment, inspect the outcome, and continue acting. Functional correctness remains behavioral: a plausible completion can preserve the requested signature yet fail on a boundary case, exceptional input, or single operator. HumanEval operationalized this distinction through executable checks [1]. Retrieval-summary integration for code intelligence complements generation by helping a system first locate and then explain candidate functions [2]. Broader evaluations exposed the gap between surface plausibility and test behavior across program-synthesis prompts [3] and APPS problems [4]. Deployment research likewise shows that model capability must be paired with an execution architecture that manages latency, cost, and resource placement [5]. Competition-level candidate sampling [6] and open multi-turn code generation [7] further demonstrate why generated programs require behavioral verification. These findings motivate treating the first program as an intermediate state rather than a final answer.

Execution has already proved useful for selecting among generated programs. CodeT couples generated tests with candidate agreement [8], while LLM-driven CI diagnosis connects workflow logs to patch recommendation [9]. LEVER learns to verify programs from specifications, source code, and execution results [10], and AIOps fault

analysis frames telemetry as evidence for diagnosis [11]. Self-Debugging feeds execution observations back into a previous program [12]. HDFS anomaly narratives with selective refusal illustrate evidence retention and abstention [13], while Reflexion stores verbalized task feedback across trials [14]. Retrieved normal prototypes support OpenStack failure analysis [15], Self-Refine repeatedly critiques and edits an output [16], and drift monitoring with safe rollback limits harmful adaptation [17]. ReAct interleaves reasoning with external actions [18], and residual fusion combines weak signals for VM degradation risk [19]. These approaches share a central concern: feedback must constrain the next action. Empirical work on code self-repair confirms that refinement without sufficiently informative external evidence is not uniformly reliable [20].

Automated program repair provides a complementary foundation. Early generate-and-validate systems searched executable patch spaces with genetic programming [21], and GenProg formalized repair through test-suite fitness [22]. SemFix translated failing behavior into repair constraints [23]. Later work established the strength of random search [24] and the benefits of condition synthesis [25], learned patch priors [26], symbolic multi-line reasoning [27], and precise predicate synthesis [28]. Surveys identify a persistent tension among search-space coverage, ranking, test adequacy, and patch overfitting [29]. Mutation testing supplies systematic fault operators

[30], while CoCoNuT [31], CURE [32], and AlphaRepair [33] use learned representations to improve proposals. Large pretrained models expand patch variety but retain the need for independent validation [34].

Patch selection is also a calibrated decision problem. Human-uncertainty distillation models disagreement instead of collapsing it to a hard label [35], while uncertainty-aware late fusion learns how confidence should shape combination rules [36]. EFR follows the same principle: a ranker score is a prior to be updated by executable evidence, not a verdict.

Verification and evidence presentation are equally important in high-stakes predictive systems. DenseNet-based cancer classification [37] and MRD prediction for multiple myeloma [38] exemplify settings where measurable evaluation must accompany a prediction. Autonomous-driving work combines perception fusion [39], attention-enhanced image detection [40], and retrieved evidence in traffic-sign explanation cards [41]. Evidence-grounded financial RAG [42], visual risk dashboards [43], long-document clause retrieval [44], progressive document rendering [45], and edge quality prediction [46] similarly prioritize decision-relevant evidence under limited attention or computation. These patterns motivate EFR's explicit separation of observations, ranking, and acceptance.

This paper connects these traditions through a model-agnostic agent grounded in executed HumanEval checks. It records assertion outcomes, exceptions, candidate-line coverage, and test-score changes, then combines them with a cross-validated patch prior. A repair counts only when the selected program passes both the feedback assertions available to the policy and the held-out assertions reserved for final verification, separating visible completion from generalization within the official checks.

The study asks how bounded iteration affects repair accuracy, which evidence sources improve ranking and efficiency, how difficulty varies by fault and symptom, and how often feedback-passing patches fail held-out checks. It evaluates all 164 tasks, six fault families, eleven configurations, budgets from one to ten, and paired task-level statistics. Figure 1 summarizes the loop.

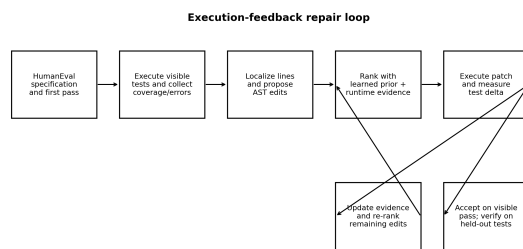


Fig. 1. Execution-feedback repair architecture. Visible tests produce structured evidence; held-out checks are used only after a candidate is selected.

METHOD

A. Benchmark Construction and Test Partitioning

HumanEval contains 164 hand-written programming tasks, each with a natural-language specification, a Python function signature, a canonical implementation, and an executable check function [1]. All 164 tasks were retained.

The canonical source for each task was normalized through Python's abstract syntax tree and executed against the complete official check function. Every canonical implementation passed before any repair trial was created. This validation step prevented harness faults from being misclassified as repair failures.

Each official check was instrumented at the assertion level, including repeated assertions inside loops. Across the benchmark, 1,582 dynamic assertions were observed (median seven; range one to 105). A deterministic hash seeded with 2023 assigned about 80% of each task's assertions to feedback and the remainder to held-out verification while preserving at least one feedback assertion for multi-check tasks. The split contained 1,260 feedback and 322 held-out assertions. Policies saw only feedback assertions; final success required the complete official stream.

One reversible first-pass fault was created per task by enumerating single-node AST edits and selecting a failing edit whose inverse remained one edit away. Six families were used: arithmetic, comparison, Boolean, constant, call, and condition. Cycling target families reduced concentration in common operators, yielding 27, 30, 11, 35, 24, and 37 faults, respectively. Every program failed the feedback and complete checks. Before repair, programs passed 32.06% of feedback assertions and 32.44% of all assertions on average. Tables I and II summarize the corpus and proposals.

Reversibility provided a known local path without revealing a task-specific template and enabled proposal-recall analysis. The exact inverse was present for 143 of 164 tasks (87.20%); behaviorally equivalent edits could still succeed when it was absent. Missing inverses therefore expose proposal limitations rather than ranking errors.

TABLE I
HUMANEVAL EXECUTION CORPUS

Item	Value	Detail
HumanEval tasks	164	All tasks
Controlled first-pass faults	164	One reversible fault per task
Dynamic official assertions	1582	Median 7; range 1-105
Feedback assertions	1260	79.6%
Held-out assertions	322	20.4%
Evaluated patch proposals	1470	Mean 8.96 per task
Canonical inverse in bounded proposal set	143	87.20% of tasks

TABLE II
FAULT AND PROPOSAL-SPACE DISTRIBUTION

Fault family	Tasks	Share (%)	Mean raw neighbors	Mean proposed	Canonical coverage (%)
Arithmetic	27	16.5	13.85	9.11	92.6
Comparison	30	18.3	13.03	9.00	96.7
Boolean	11	6.7	24.36	11.91	54.5
Constant	35	21.3	16.71	9.06	77.1
Call	24	14.6	16.79	10.00	87.5
Condition	37	22.6	11.51	7.19	94.6

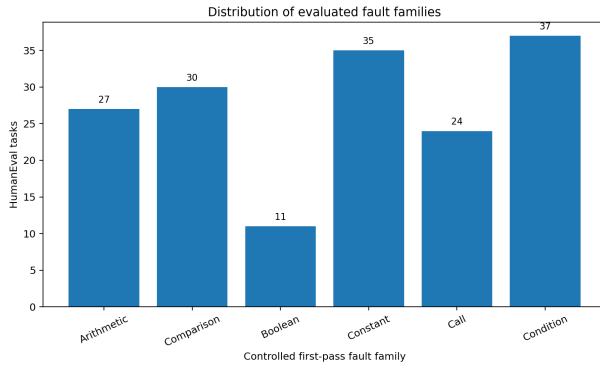


Fig. 2. Distribution of the 164 controlled first-pass faults across six edit families.

B. Execution Harness and Runtime Evidence

Each candidate ran in a fresh Python namespace. The harness recorded assertion outcomes, exception details, executed source lines, and exception locations. A hard whole-check timeout marked nonterminating or excessively slow candidates as failures, including calls made during test setup. This covered checks that invoke the candidate inside loops before asserting on stored results.

Coverage was collected only for the initial faulty program on feedback tests. With F failing assertions, $ef(l)$ failing assertions executing line l , and $ep(l)$ passing assertions executing it , suspiciousness was $O(l) = ef(l) / \sqrt{F[ef(l) + ep(l)]}$. Ochiai performs strongly for fault localization [47], while Tarantula established passing/failing coverage contrast [48]. Exception lines received suspiciousness at least one, and the dominant exception type added a small edit-family compatibility score.

The feedback record is deliberately more structured than a free-form diagnostic message. It contains evidence that can be consumed by a deterministic policy, learned ranker, or language-model planner. The same separation appears in operational analytics: deep-autoencoder IoT analysis converts traffic into anomaly signals [49], DDoS mitigation maps monitored features to bounded interventions [50], and C5.0 credit-risk modeling preserves an auditable decision path [51]. Macro-market uncertainty fusion [52], AI-driven customer risk analysis [53], and enterprise marketing prediction [54] likewise keep observable inputs distinct from downstream decisions. In EFR, assertion outcomes and coverage play that intermediate role, keeping each decision trace reproducible from saved candidate evaluations and attempt logs.

C. Localized Patch Proposal Space

The generator enumerated one-edit AST neighbors under the six legal families but received neither the original mutation location nor inverse. Each proposal stored its family, line, node context, transition, source-length change, and operation description. It retained edits on the two most suspicious lines plus globally strong alternatives, capped at 12 per task; smaller legal sets were kept in full.

The cap reduced 2,447 raw neighbors to 1,470 executed proposals (8.96 per task); the raw space averaged 14.92 neighbors and initial coverage touched 5.35 lines. Because execution has cost, the design separates proposal recall from ranking: a valid patch must enter the bounded set and be reached before the budget expires. Defects4J likewise separates benchmark construction, test evidence, and repair evaluation [55].

D. Cross-Validated Patch Prior

A class-balanced logistic model estimated complete-check success from edit family, node and parent type, transition, normalized line position, source-length change, and operator prior. Five-fold group cross-validation used task identifiers as groups and a fixed seed. Each task's probabilities therefore came from a model trained only on proposals from other tasks.

The ranker is intentionally compact. Its purpose is to test whether transferable structural regularities improve search, not to reproduce a large neural code model. Spectral rank aggregation formalizes recovery of an ordering from noisy comparisons [56]. E-commerce recommendation provides a practical ranking setting [57], counterfactual learning-to-rank emphasizes evaluation from logged feedback [58], and uncertainty quantification for spectral estimators cautions against treating a point score as certainty [59]. EFR therefore uses the learned probability as an empirical prior over patch shapes and delegates acceptance to execution. The same interface could receive a prior from a code model, retrieval system, or verifier without changing the execution-feedback loop.

E. Repair Policies

Table III defines the policies. One-shot baselines selected by static prior or cross-validated probability. Random search executed up to five deterministically shuffled proposals; coverage search combined Ochiai, exception compatibility, and a static prior; learned search used fixed probability order. No policy retained the failing first pass. EFR combined learned and runtime signals. Candidate c on line lc received $B(c) = 3.0pL(c) + 0.5O(lc) + 0.2E(c) + 0.05S(c)$, using learned probability, Ochiai, exception compatibility, and static prior. After attempt t , positive feedback-score changes accumulated as line evidence $Dt(l)$, while non-improving attempts accumulated by family $Nt(g)$. Remaining proposals were scored $Bt(c) = B(c) + Dt(lc) - 0.15Nt(gc)$, rewarding improving lines and modestly discouraging repeatedly unsuccessful families.

A policy stopped when a candidate passed all feedback assertions, then ran the complete stream including held-out checks. If none passed within budget, the last candidate became the final state. EFR used budgets of five and ten; ten-attempt ablations removed dynamic re-ranking, coverage, or the learned prior. Identical proposal executions were reused across policies.

Table III Repair Policy Configurations

Method	Budget	Runtime evidence	Selection policy
No repair	0	None	Initial program retained
Static heuristic	1	AST prior	Highest fixed operator prior
Learned ranker	1	Cross-validated patch prior	Highest learned probability
Random search	5	Binary test result	Seeded random proposal order
Coverage search	5	Coverage, exception type	Ochiai localization plus error compatibility
Learned search	5	Cross-validated patch prior	Top-ranked proposals in fixed order
EFR agent	5	Coverage, exception, test delta, learned prior	Re-rank after each failed execution

Method	Budget	Runtime evidence	Selection policy
EFR agent	10	Coverage, exception, test delta, learned prior	Same policy with larger attempt budget

F. Metrics and Statistical Analysis

Repair Pass@1 is the fraction repaired by the first proposal; final repair requires all official checks. Mean attempts, execution time, and lexical context quantify cost. The context metric counts the specification, current source, structured feedback, and attempted-patch descriptions available at each decision. Feedback-pass and held-out-pass counts measure visible completion and overfitting. Wilson intervals quantify uncertainty around repair proportions. Because every method was evaluated on the same 164 tasks, paired differences were estimated with a 20,000-replicate task bootstrap. Exact McNemar tests compared EFR with each baseline using discordant task outcomes, and Holm correction controlled family-wise error across six comparisons. These analyses use the task, rather than an assertion or proposal, as the statistical unit.

RESULTS AND DISCUSSION

A. Overall Repair Effectiveness

Table IV and Figure 3 give the main comparison. Static heuristic repaired 26.22% and the one-shot learned ranker 57.32%. Random search also reached 57.32% but averaged 3.51 attempts; coverage search reached 66.46%, and learned fixed-order search 82.32%. Proposal ordering therefore mattered more than undirected iteration.

EFR(5) repaired 137 of 164 tasks (83.54%) with 59.76% Pass@1. EFR(10) repaired 144 (87.80%; 95% Wilson interval, 81.91%-91.97%) while averaging 2.77 attempts, 85.9 ms, and 914.6 lexical tokens. Its median was one because the first proposal repaired 98 tasks.

EFR's ten-attempt result exceeded the 87.20% rate at which the exact inverse mutation appeared in the bounded proposal set. This is not a contradiction: at least one task was solved by a distinct proposal that was behaviorally equivalent under every official assertion. The observation is important for code agents because source-level identity is stricter than functional repair. The evaluation consistently uses complete-test behavior as the acceptance criterion.

Paired tests in Table X confirm the gains. EFR(10) added seven repairs over EFR(5) and nine over learned search, with no losses; Holm-adjusted p-values were 0.0156 and 0.0078, respectively. The learned-search difference was 5.49 points with a 95% paired-bootstrap interval of 2.44-9.15 points. Larger advantages over weaker baselines also remained significant.

Table IV Overall Repair Effectiveness And Cost

Method	Pass@1 (%)	Final (%)	95% CI	Feedback (%)	Attempts	Lexical tokens	Exec. ms
No repair	0.00	0.00	[0.00, 2.29]	0.00	0.00	289.5	0.0
Static heuristic (1)	26.22	26.22	[20.09, 33.44]	26.83	1.00	480.8	4.6
Learned ranker (1)	57.32	57.32	[49.66, 64.63]	57.93	1.00	480.9	6.3
Random search (5)	20.73	57.32	[49.66, 64.63]	58.54	3.51	1009.5	67.1

Method	Pass@1 (%)	Final (%)	95% CI	Feedback (%)	Attempts	Lexical tokens	Exec. ms
Coverage search (5)	33.54	66.46	[58.93, 73.24]	67.07	3.09	936.8	63.2
Learned search (5)	57.32	82.32	[75.76, 87.40]	82.93	2.13	731.5	24.9
EFR agent (5)	59.76	83.54	[77.11, 88.43]	84.15	2.09	739.0	53.3
EFR agent (10)	59.76	87.80	[81.91, 91.97]	88.41	2.77	914.6	85.9

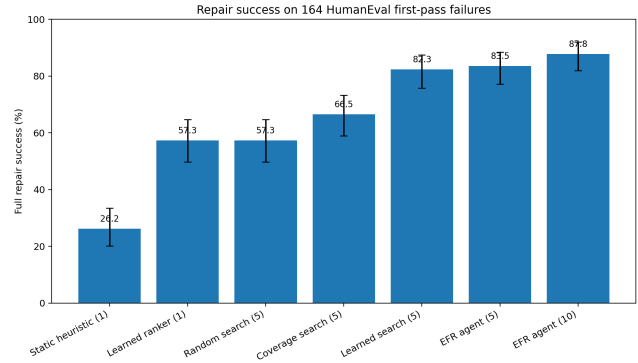


Fig. 3. Full repair success with 95% Wilson intervals. Every method was evaluated on the same 164 tasks.

B. Attempt Budget and Diminishing Returns

Table V and Figure 4 trace the same EFR policy as its maximum budget increases. One attempt solved 59.76% of tasks. A second attempt raised success to 69.51%, and a third raised it to 78.05%. Five attempts reached 83.54%, seven reached 86.59%, and ten reached 87.80%. The first three attempts therefore delivered 78.05 of the final 87.80 percentage points, while attempts eight through ten added only 1.22 points beyond the seven-attempt result.

Early stopping made cost sublinear: increasing the budget from five to ten raised mean attempts from 2.09 to 2.77 and tokens from 739.0 to 914.6. EFR(5) and learned search suit cost-sensitive settings, while EFR(10) supplies seven additional verified repairs. A larger maximum budget therefore need not impose its full cost on easy tasks.

Table V Efr Attempt-Budget Sensitivity

Maximum attempts	Full repair (%)	Feedback pass (%)	Mean attempts used	Mean lexical tokens
1	59.76	60.37	1.00	496.0
2	69.51	70.12	1.40	578.7
3	78.05	78.66	1.70	644.6
5	83.54	84.15	2.09	739.0
7	86.59	87.20	2.39	816.3
10	87.80	88.41	2.77	914.6

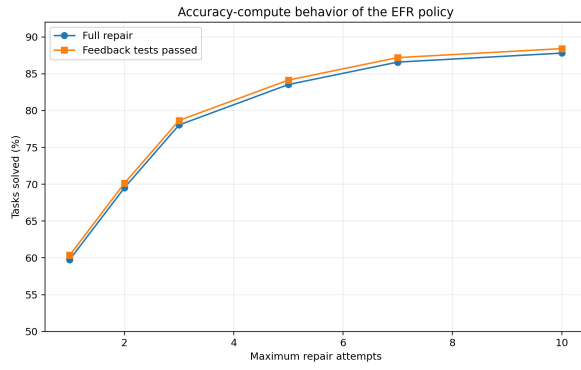


Fig. 4. EFR success as the maximum repair budget grows from one to ten attempts.

C. Fault-Family Behavior

Table VI and Figure 5 reveal substantial heterogeneity across fault families. EFR(10) repaired 96.67% of comparison faults and 97.30% of condition faults. These categories often produce sharply contrasting passing and failing paths, which gives coverage localization a useful line-level signal. Arithmetic repairs reached 92.59%, and call repairs reached 87.50%. The five-attempt policy already solved 88.89% of arithmetic faults and all call faults that were eventually solved, indicating that the learned prior placed many call corrections early.

Constant faults benefited most from additional search, rising from 57.14% at five attempts to 74.29% at ten. Boolean faults were hardest at 63.64%, and their exact inverse appeared in only 54.55% of bounded sets versus more than 92% for arithmetic, comparison, and condition faults. This points to proposal recall: expanding or learning Boolean edits is more promising than reordering the same 12 candidates.

Coverage search was particularly strong on comparison and condition faults but weak on Boolean and call faults. This pattern explains why coverage alone reached 66.46% overall yet trailed learned search by 15.85 points. Spectrum localization identifies where behavior diverges; it does not necessarily identify which syntactic transition is appropriate. The learned prior supplies that complementary preference.

Table Vi Full Repair Success By Fault Family (%)

Fault family	Tasks	Coverage-5	EFR-10	EFR-5	Learned-5	Random-5
arithmetic	27	51.9	92.6	88.9	85.2	55.6
comparison	30	93.3	96.7	96.7	96.7	66.7
boolean	11	27.3	63.6	63.6	63.6	27.3
constant	35	71.4	74.3	57.1	54.3	54.3
call	24	33.3	87.5	87.5	87.5	41.7
condition	37	83.8	97.3	97.3	97.3	73.0

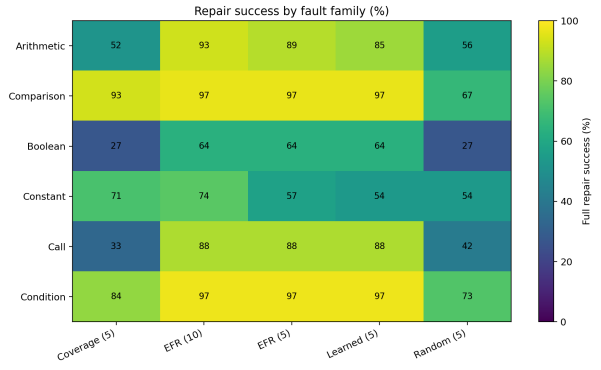


Fig. 5. Repair success by fault family and policy. Values are percentages of tasks in each family.

D. Failure Symptoms and Diagnostic Value

AssertionFailure dominated 145 tasks; EFR(10) repaired 86.21%, versus 81.38% for learned search and 64.83% for coverage search. All eight TypeError tasks were repaired by learned search and both EFR budgets, and the eleven pooled IndexError, ZeroDivisionError, ValueError, AttributeError, and timeout tasks reached 100% with EFR(10) (Table VII).

Exception compatibility is most useful when runtime type or arithmetic information narrows the edit family. Agents should therefore retain exception classes and locations rather than collapse execution to a binary reward. The small groups limit generalization: the 100% values cover eleven pooled tasks and eight TypeError tasks.

Table Vii
Full Repair Success By Initial Failure Symptom (%)

Initial symptom	Tasks	Coverage-5	EFR-10	EFR-5	Learned-5
AssertionFailure	145	64.8	86.2	82.1	81.4
Other exception	11	81.8	100.0	90.9	81.8
TypeError	8	75.0	100.0	100.0	100.0

E. Evidence Ablations

Table VIII separates early ranking from final budget saturation. Removing dynamic re-ranking left final ten-attempt accuracy unchanged at 87.80% and changed mean attempts only from 2.77 to 2.76. Once ten of at most twelve proposals can be tried, the fixed base ranking already reaches nearly every solvable candidate. Dynamic updates are therefore not the source of the final ten-attempt gain.

Removing coverage lowered Pass@1 from 59.76% to 56.71% and raised mean attempts from 2.77 to 2.86, but final accuracy stayed 87.80%. Removing the learned prior reduced Pass@1 to 32.93%, increased attempts to 3.88 and tokens to 1,139.8, and lowered final repair to 86.59%. The prior drives efficiency; execution evidence supports localization, stopping, and recovery.

At five attempts, dynamic EFR repaired 83.54% versus 82.32% for fixed learned search at similar cost. Feedback therefore helps most before the budget approaches enumeration. On this bounded single-edit benchmark, repeated execution is useful, but dynamic re-ranking becomes less important when nearly all proposals can be tried.

Table VIII Ten-Attempt Efr Ablations

Configuration	Repair Pass@1 (%)	Final repair (%)	Mean attempts	Mean lexical tokens	Mean execution ms
EFR agent (10)	59.76	87.80	2.77	914.6	85.9
EFR no dynamic (10)	59.76	87.80	2.76	910.5	82.4
EFR no coverage (10)	56.71	87.80	2.86	931.5	85.9
EFR no learned prior (10)	32.93	86.59	3.88	1139.8	91.5

F. Feedback-Test Overfitting

Table IX compares visible completion with complete verification. Learned search produced 136 feedback-passing patches, of which 135 passed the held-out assertions. EFR(5) produced 138 feedback-passing patches and 137 full repairs. EFR(10) produced 145 feedback-passing patches and 144 full repairs. In each method, exactly one visible-test success failed held-out verification, corresponding to conditional generalization rates from 99.26% to 99.31%.

Only one feedback-passing patch failed held-out verification. The low rate reflects the local search space and dense official checks, not a general claim that visible tests suffice. Without the held-out gate, three methods would each receive one additional incomplete repair.

Table IX Feedback-To-Held-Out Generalization

Method	Feedback not passed	Feedback pass / held-out fail	Full repair	Generalization among feedback passes (%)
Learned search (5)	28	1	135	99.26
EFR agent (5)	26	1	137	99.28
EFR agent (10)	19	1	144	99.31

G. Effectiveness-Cost Tradeoffs and Agent Design

Figure 6 places repair success against lexical context cost. The one-shot learned ranker is inexpensive but leaves 70 tasks unrepaired. Random search consumes the most context among the principal baselines while matching the learned ranker's accuracy. Coverage search improves accuracy but remains less efficient than learned search. EFR(5) adds 1.22 points over learned search for only 7.5 additional mean tokens, while EFR(10) adds a further 4.27 points for 175.6 additional tokens. These points define practical choices rather than a single universally optimal budget. Comparable finite-budget decisions appear in risk-aware auto-bidding [60], constrained marketing allocation [61], and digital-twin dispatch [62]: each allocates limited resources to the highest-value actions rather than exploring indefinitely.

Three design implications follow. Separate proposal generation, ranking, and verification; return structured per-test outcomes, exceptions, and coverage after execution; and bound iteration with verified stopping. The median of one attempt and flattening after seven show that fixed long dialogues waste computation. Agents should spend effort only while executable evidence supports another edit.

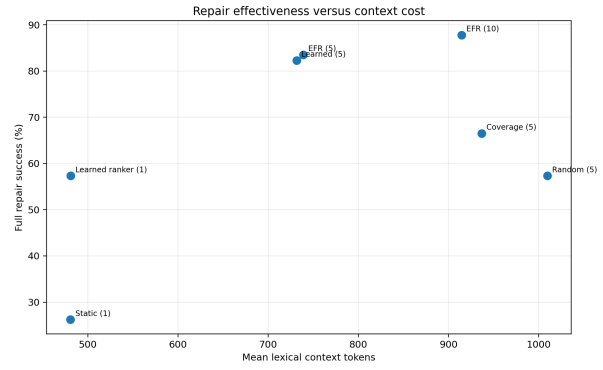


Fig. 6. Full repair success versus mean lexical context cost for the principal repair methods.

Table X Paired Comparisons Against Efr(10)

Comparator	Difference (pp)	95% paired CI	EFR-only	Comparator-only	Holm p
EFR agent (5)	4.27	[1.22, 7.32]	7	0	0.0156
Learned search (5)	5.49	[2.44, 9.15]	9	0	0.00781
Coverage search (5)	21.34	[15.24, 28.05]	35	0	1.75e-10
Random search (5)	30.49	[23.17, 37.80]	50	0	8.88e-15
Learned ranker (1)	30.49	[23.78, 37.80]	50	0	8.88e-15
Static heuristic (1)	61.59	[54.27, 68.90]	101	0	4.73e-30

IV. LIMITATIONS

Each first-pass program contains one controlled AST edit to a HumanEval canonical solution. Real outputs may contain interacting defects, missing algorithms, wrong interfaces, fabricated dependencies, or misunderstood specifications. The reported rates therefore measure local functional repair, not end-to-end software engineering. HumanEval comprises short, self-contained Python functions and omits repositories, builds, services, concurrency, APIs, and cross-file changes. Feedback and held-out assertions originate from the same official checks, so they can share input logic even though policies cannot see the held-out partition. Independently authored tests may reveal more overfitting.

The exact inverse appeared in 87.20% of bounded sets and less often for Boolean and constant faults, so proposal recall limits the ceiling. Other localization rules, larger caps, or model-generated edits would change it. Legal edits were single-node transformations, and passing official checks established behavioral adequacy only within those tests.

The learned prior and weights were developed on HumanEval-derived instances. Grouped cross-validation prevents target-task label leakage, but features and weights may remain benchmark-specific; broader transfer requires MBPP, APPS, Defects4J, and repository-level evaluation. Approximately shared-feature learning suggests transfer depends on partial source-target structure [63]. Educational use also depends on feedback language and interaction design, as bilingual CS instruction affects engagement and learning [64]. Lexical tokens and runtime are most reliable for within-experiment comparison.

Production execution needs stronger isolation than this benchmark harness: containers or micro-VMs, quotas,

filesystem and network restrictions, dependency allowlists, and audit logs. Time limits and fresh namespaces were sufficient here but are not a general security boundary.

V. CONCLUSION

Across all 164 HumanEval tasks, EFR executed 1,470 proposals against 1,582 assertions, exposed 1,260 to repair policies, and reserved 322 for final verification. A cross-validated structural prior drove early efficiency; coverage, exceptions, and test deltas supplied runtime evidence.

EFR repaired 83.54% within five attempts and 87.80% within ten, versus 82.32% for learned search, 66.46% for coverage search, and 57.32% for random search. EFR(10) averaged 2.77 attempts, significantly outperformed learned search, and produced only one feedback-passing patch that failed held-out checks. Most gains arrived by attempt five; later trials addressed ambiguous constants and long-tail rankings.

Reliable repair follows a disciplined loop: execute, preserve structured evidence, localize, generate bounded edits, rank with transferable priors, and verify before acceptance. The loop can wrap deterministic repair or an LLM agent, but executable behavior remains the arbiter of correctness.

REFERENCES

- [1] M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.
- [2] Y. Li, "Findable then Explainable: Retrieval-Summary Integration for Code Intelligence on a Lightweight CodeSearchNet Subset," *J. Adv. Comput. Syst.*, vol. 4, no. 7, pp. 65-82, Jul. 2024, doi: 10.69987/JACS.2024.40706.
- [3] J. Austin et al., "Program Synthesis with Large Language Models," arXiv:2108.07732, 2021.
- [4] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, "Measuring Coding Challenge Competence With APPS," in *Advances in Neural Information Processing Systems*, vol. 34, 2021, pp. 12621-12632.
- [5] Q. Xin, "Hybrid Cloud Architecture for Efficient and Cost-Effective Large Language Model Deployment," *J. Inf. Syst. Informatics*, vol. 7, no. 3, pp. 2182-2195, Sep. 2025, doi: 10.51519/journalisi.v7i3.1170.
- [6] Y. Li et al., "Competition-Level Code Generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092-1097, 2022.
- [7] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis," in *Proc. International Conference on Learning Representations*, 2023.
- [8] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "CodeT5: Code Generation with Generated Tests," arXiv:2207.10397, 2022.
- [9] H. Zhang, "LLM-Driven CI Failure Diagnosis and Automated Repair: From GitHub Actions Logs to Patch Recommendation," *J. Technol. Informatics Eng.*, vol. 4, no. 1, pp. 190-214, Apr. 2025, doi: 10.51903/jtie.v4i1.484.
- [10] Ni, S. Iyer, D. Radev, V. Stoyanov, W.-T. Yih, S. Wang, and X. V. Lin, "LEVER: Learning to Verify Language-to-Code Generation with Execution," in *Proc. 40th International Conference on Machine Learning*, PMLR, vol. 202, 2023, pp. 26106-26128.
- [11] Y. He, Y. Pan, Y. Wang, S. Du, and Q. Xin, "Intelligent Fault Analysis with AIOps Technology," *J. Theory Pract. Eng. Sci.*, vol. 4, no. 1, pp. 94-100, Feb. 2024, doi: 10.53469/jtpes.2024.04(01).13.
- [12] X. Chen, M. Lin, N. Scharli, and D. Zhou, "Teaching Large Language Models to Self-Debug," arXiv:2304.05128, 2023.
- [13] J. Nie and D. Zheng, "Ambiguity-Aware HDFS Log Anomaly Detection with Retrieval-Augmented Failure Narratives and Selective Refusal," *J. Adv. Comput. Syst.*, vol. 3, no. 1, pp. 66-80, Jan. 2023, doi: 10.69987/JACS.2023.30105.
- [14] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language Agents with Verbal Reinforcement Learning," in *Advances in Neural Information Processing Systems*, vol. 36, 2023, pp. 8634-8652.
- [15] Q. Xin, "Explaining OpenStack Failure-Injection Log Anomalies with Retrieved Normal Prototypes," *Emerg. Inf. Sci. Technol.*, vol. 6, no. 2, pp. 125-146, Nov. 2025, doi: 10.18196/eist.v6i2.31232.
- [16] Madaan et al., "Self-Refine: Iterative Refinement with Self-Feedback," in *Advances in Neural Information Processing Systems*, vol. 36, 2023, pp. 46534-46594.
- [17] H. Zhang, "DriftGuard: Multi-Signal Drift Early Warning and Safe Re-Training/Rollback for CTR/CVR Models," *J. Adv. Comput. Syst.*, vol. 3, no. 7, pp. 24-40, Jul. 2023, doi: 10.69987/JACS.2023.30703.
- [18] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafraan, K. Narasimhan, and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," in *Proc. International Conference on Learning Representations*, 2023.
- [19] J. Nie and D. Zheng, "Noisy-Neighbor-Aware VM Degradation Risk Modeling with Unsupervised Residual Fusion," *J. Adv. Comput. Syst.*, vol. 4, no. 4, pp. 112-123, Apr. 2024, doi: 10.69987/JACS.2024.40409.
- [20] T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama, "Is Self-Repair a Silver Bullet for Code Generation?" arXiv:2306.09896, 2023.
- [21] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically Finding Patches Using Genetic Programming," in *Proc. 31st International Conference on Software Engineering*, 2009, pp. 364-374.
- [22] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for \$8 Each," in *Proc. 34th International Conference on Software Engineering*, 2012, pp. 3-13.
- [23] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program Repair via Semantic Analysis," in *Proc. 35th International Conference on Software Engineering*, 2013, pp. 772-781.
- [24] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The Strength of Random Search on Automated Program Repair," in *Proc. 36th International Conference on Software Engineering*, 2014, pp. 254-265.
- [25] F. Long and M. Rinard, "Staged Program Repair with Condition Synthesis," in *Proc. 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 166-178.
- [26] F. Long and M. Rinard, "Automatic Patch Generation by Learning Correct Code," in *Proc. 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298-312.
- [27] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis," in *Proc. 38th International Conference on Software Engineering*, 2016, pp. 691-701.
- [28] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise Condition Synthesis for Program Repair," in *Proc. 39th International Conference on Software Engineering*, 2017, pp. 416-426.
- [29] M. Monperrus, "Automatic Software Repair: A Bibliography," *ACM Computing Surveys*, vol. 51, no. 1, Art. no. 17, 2018.
- [30] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation Testing Advances: An Analysis and Survey," *Advances in Computers*, vol. 112, pp. 275-378, 2019.
- [31] T. Lutellier, H. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair," in *Proc. 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 101-114.
- [32] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-Aware Neural Machine Translation for Automatic Program Repair," in *Proc. 43rd IEEE/ACM International Conference on Software Engineering*, 2021, pp. 1161-1173.
- [33] C. S. Xia and L. Zhang, "Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning," in *Proc. 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 959-971.
- [34] C. S. Xia, Y. Wei, and L. Zhang, "Automated Program Repair in the Era of Large Pre-Trained Language Models," in *Proc. 45th IEEE/ACM International Conference on Software Engineering*, 2023, pp. 1482-1494.
- [35] Z. S. Zhong, R. Ma, and H. Zhao, "Human-Uncertainty Distillation for Calibrated Vision Models on CIFAR-10H," *J. Adv. Comput. Syst.*, vol. 3, no. 2, pp. 77-89, Feb. 2023, doi: 10.69987/JACS.2023.30206.
- [36] Q. Xin, "Uncertainty-Aware Late Fusion for 3D Perception (Confidence Calibration + Fusion Rule Learning)," *J. Technol. Informatics Eng.*, vol. 4, no. 1, pp. 215-238, Apr. 2025, doi: 10.51903/jtie.v4i1.485.
- [37] Z. Zhong, M. Zheng, H. Mai, J. Zhao, and X. Liu, "Cancer Image Classification Based on DenseNet Model," *J. Phys.: Conf. Ser.*, vol. 1651, no. 1, Art. no. 012143, Nov. 2020, doi: 10.1088/1742-6596/1651/1/012143.
- [38] J. Chen, J. Xiong, Y. Wang, Q. Xin, and H. Zhou, "Implementation of an AI-Based MRD Evaluation and Prediction Model for Multiple

- Myeloma," *Front. Comput. Intell. Syst.*, vol. 6, no. 3, pp. 127-131, Jan. 2024, doi: 10.54097/zj4MnbWW.
- [39] Y. Wang, S. Du, Q. Xin, Y. He, and W. Qian, "Autonomous Driving System Driven by Artificial Intelligence Perception Fusion," *Acad. J. Sci. Technol.*, vol. 9, no. 2, pp. 193-198, Feb. 2024, doi: 10.54097/e0b9ak47.
- [40] Z. Ling, Q. Xin, Y. Lin, G. Su, and Z. Shui, "Optimization of Autonomous Driving Image Detection Based on RFACnv and Triplet Attention," *Appl. Comput. Eng.*, vol. 77, no. 1, pp. 210-217, 2024, doi: 10.54254/2755-2721/77/2024MA0067.
- [41] R. Ma, L. Zhang, and T. Song, "Computer-Vision-Informed Visual Explanation Cards for Autonomous-Driving Traffic-Sign Alerts: Localization, Classification, and Retrieved Evidence on GTSDB," *Int. J. Graph. Des.*, vol. 3, no. 2, pp. 455-473, Oct. 2025, doi: 10.51903/ijgd.v3i2.3991.
- [42] S. Zhou, Z. Li, and E. Wang, "Evidence-Grounded RAG for Tokenized Trade Receivable Disclosure QA under U.S. Capital Market Standards," *J. Adv. Comput. Syst.*, vol. 3, no. 7, pp. 41-57, Jul. 2023, doi: 10.69987/JACS.2023.30704.
- [43] Z. Li, S. Zhou, and Z. Zhou, "Financial Risk Dashboard Design for Institutional RWA Investors: Visual Hierarchy, Chart Comprehension, and Explainability in FinChart-Bench," *Int. J. Graph. Des.*, vol. 3, no. 1, pp. 196-210, May 2025, doi: 10.51903/ijgd.v3i1.3715.
- [44] S. Zhou, Z. Li, and E. Wang, "Long-Document RAG for Contractual and Insurance Clause Analysis in Receivables RWA Structures," *J. Adv. Comput. Syst.*, vol. 4, no. 8, pp. 88-104, Aug. 2024, doi: 10.69987/JACS.2024.40810.
- [45] H. Wang, Y. Ren, and X. Chang, "Layout-Aware Progressive PDF Rendering: AI Prioritization of PDF Slices to Reduce Time-to-Functional-First-Frame on FUNSD," *J. Technol. Informatics Eng.*, vol. 4, no. 2, pp. 425-446, Aug. 2025, doi: 10.51903/jtie.v4i2.523.
- [46] B. Zhou, H. Wang, and X. Chang, "Distilling VMAF into an Edge-Deployable Quality Predictor: A Pilot Shot-Level Proxy with LLM-Ready Quality Tokens," *J. Technol. Informatics Eng.*, vol. 4, no. 2, pp. 447-463, Aug. 2025, doi: 10.51903/jtie.v4i2.522.
- [47] R. Abreu, P. Zoetewij, and A. J. C. van Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization," in *Proc. 12th Pacific Rim International Symposium on Dependable Computing*, 2006, pp. 39-46.
- [48] J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," in *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005, pp. 273-282.
- [49] Q. Xin, Z. Xu, L. Guo, F. Zhao, and B. Wu, "IoT Traffic Classification and Anomaly Detection Method Based on Deep Autoencoders," *Appl. Comput. Eng.*, vol. 69, no. 1, pp. 64-70, Jul. 2024, doi: 10.54254/2755-2721/69/20241511.
- [50] B. Wang, Y. He, Z. Shui, Q. Xin, and H. Lei, "Predictive Optimization of DDoS Attack Mitigation in Distributed Systems Using Machine Learning," *Appl. Comput. Eng.*, vol. 64, no. 1, pp. 89-94, 2024, doi: 10.54254/2755-2721/64/20241350.
- [51] Q. Xin, R. Song, Z. Wang, Z. Xu, and F. Zhao, "Enhancing Bank Credit Risk Management Using the C5.0 Decision Tree Algorithm," *J. Comput. Technol. Appl. Math.*, vol. 1, no. 4, pp. 100-107, Nov. 2024, doi: 10.5281/zenodo.14032041.
- [52] H. Zhou and K. Zhang, "News-Based Uncertainty and Macro-Market Fusion for VIX Direction Forecasting: Evidence from 2015-2024 FRED Panel," *J. Technol. Informatics Eng.*, vol. 4, no. 2, pp. 487-501, Aug. 2025, doi: 10.51903/jtie.v4i2.540.
- [53] T. Yang, Q. Xin, X. Zhan, S. Zhuang, and H. Li, "Enhancing Financial Services through Big Data and AI-Driven Customer Insights and Risk Analysis," *J. Knowl. Learn. Sci. Technol.*, vol. 3, no. 3, pp. 53-62, Jul. 2024, doi: 10.60087/jklst.vol3.n3.p53-62.
- [54] J. Wang, Q. Xin, Y. Liu, J. Wang, and T. Yang, "Predicting Enterprise Marketing Decision Making with Intelligent Data-Driven Approaches," *J. Ind. Eng. Appl. Sci.*, vol. 2, no. 3, pp. 12-19, Jun. 2024, doi: 10.5281/zenodo.11357252.
- [55] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," in *Proc. International Symposium on Software Testing and Analysis*, 2014, pp. 437-440.
- [56] Z. S. Zhong and S. Ling, "Improved Theoretical Guarantee for Rank Aggregation via Spectral Method," *Inf. Inference: J. IMA*, vol. 13, no. 3, Art. no. iaae020, Sep. 2024, doi: 10.1093/imaia/iaae020.
- [57] K. Xu, H. Zhou, H. Zheng, M. Zhu, and Q. Xin, "Intelligent Classification and Personalized Recommendation of E-Commerce Products Based on Machine Learning," *Appl. Comput. Eng.*, vol. 64, no. 1, pp. 143-149, 2024, doi: 10.54254/2755-2721/64/20241365.
- [58] H. Zhang, "Counterfactual Learning-to-Rank for Ads: Off-Policy Evaluation on the Open Bandit Dataset," *J. Adv. Comput. Syst.*, vol. 5, no. 12, pp. 1-11, Dec. 2025, doi: 10.69987/JACS.2025.51201.
- [59] Z. S. Zhong and S. Ling, "Uncertainty Quantification of Spectral Estimator and MLE for Orthogonal Group Synchronization," *arXiv:2408.05944*, Aug. 2024, doi: 10.48550/arXiv.2408.05944.
- [60] H. Zhang, "Risk-Aware Budget-Constrained Auto-Bidding under First-Price RTB: A Distributional Constrained Deep Reinforcement Learning Framework," *J. Adv. Comput. Syst.*, vol. 4, no. 6, pp. 30-47, Jun. 2024, doi: 10.69987/JACS.2024.40603.
- [61] Y. Lu, H. Zhou, and Y. Zhang, "A Constrained, Data-Driven Budgeting Framework Integrating Macro Demand Forecasting and Marketing Response Modeling," *J. Technol. Informatics Eng.*, vol. 4, no. 3, pp. 493-520, Dec. 2025, doi: 10.51903/jtie.v4i3.466.
- [62] L. Zhang, R. Ma, and P. Greg, "Digital-Twin Dispatching for Urban Mobility via Spatio-Temporal Transformers and Offline Reinforcement Learning," *J. Technol. Informatics Eng.*, vol. 4, no. 2, pp. 337-363, Aug. 2025, doi: 10.51903/jtie.v4i2.501.
- [63] Z. S. Zhong, X. Pan, and Q. Lei, "Bridging Domains with Approximately Shared Features," in *Proc. 28th Int. Conf. Artificial Intelligence and Statistics (AISTATS)*, PMLR, vol. 258, 2025, pp. 559-567.
- [64] G. S. Raj et al., "Impact of Bilingual CS Education on Student Learning and Engagement in a Data Structures Course," in *Proc. 19th Koli Calling Int. Conf. Comput. Educ. Res.*, 2019, pp. 1-10, doi: 10.1145/3364510.3364518.