Journal of Data Science and Modeling, Vol. 3, No. 2, 69-85, June 2025

Research Manuscript

Fuzzy-AutoRepair: Expert-Validated Repair-Pattern-Conditioned Re-Ranking and Search-Cost Analysis for C/C++ Program Repair

Amir Ghanati[1], Saeed Parsa[2]*, and Habib Izadkhah[3]

[1]Department of Computer Engineering and Information Technology, Qa.C., Islamic Azad University, Qazvin, Iran
[2]School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran
[3]Department of Computer Science, Faculty of Mathematical Sciences, University of Tabriz, Tabriz, Iran

Recieved: 05/07/2026 Accepted: 30/08/2026

Abstract: Automated program repair (APR) for C/C++ is constrained by uncertain fault localization, large candidate spaces, and costly validation. Building on SpecPatch, Fuzzy-AutoRepair introduces an expert-validated, repair-pattern-conditioned re-ranking layer before candidate generation. It maps a 40-feature representation of each suspicious region to seven linguistic evidence variables, applies a 42-rule Mamdani inference system, and fuses fuzzy suitability with learning-to-rank (LTR) relevance and structural-risk damping while preserving hard transformation constraints. On 2,244 independent samples, it improves Hit@1 from 0.45 to 0.50, macro Recall@5 from 0.806 to 0.846, and NDCG@5 from 0.78 to 0.84 relative to LTR alone. On 69 C/C++ defects, it reduces generated candidates, compilation attempts, and runtime by 32.2%, 29.4%, and 15.6%, respectively, with 1.31 ms mean fuzzy-inference overhead. On 181 eligible BUGSC++ defects, corresponding reductions are 30.8%, 28.2%, and 15.0%. Binary repair-success differences remain statistically inconclusive. The contribution is an auditable mechanism for improving repair-pattern ordering and search cost, rather than a new fault localizer or candidate generator.

Keywords: Automated program repair, C/C++, fuzzy logic, learning to rank, interpretable machine learning, repair patterns, patch ranking, program analysis, empirical software engineering.
Mathematics Subject Classification (2010): 68-04, 68N30.

*Corresponding author. Email: parsa@iust.ac.ir

1. Introduction

Generate-and-validate automated program repair (APR) constructs candidate patches and checks them against an available correctness specification, usually a regression test suite Gazzola *et al.* (2019). For C/C++ systems, the search remains expensive because fault localization is uncertain, a suspicious region may support several repair families, and each candidate may require compilation and test execution. A fixed repair-pattern order can therefore waste validation effort, while a purely learned ranking can be difficult to inspect and may mix relevance, structural risk, and transformation legality.

Fuzzy reasoning is useful at this decision point because it represents partial evidence through linguistic terms rather than forcing a crisp pattern choice. Fuzzy-AutoRepair builds on the SpecPatch framework Ghanati *et al.* (forthcoming), which supplies the suspicious-region representation, thirteen repair-pattern families, and AST-level candidate-generation and validation substrate. The present study does not change raw suspiciousness scores or introduce a new candidate generator. Instead, it interprets each suspicious region through seven fuzzy evidence variables, ranks the inherited repair families, and keeps syntax, type, scope, signature, and structural constraints as hard legality checks.

The contributions are fourfold. First, the method introduces repair-pattern-conditioned ordering for localization-uncertain APR, including propagation-aware prioritization of Neighbourhood Repair when local evidence is weak but dependence evidence is strong. Second, it formalizes a 42-rule Mamdani rule base reviewed by two APR experts and one fuzzy-systems expert. Third, it combines learned relevance, fuzzy suitability, and structural-risk damping in a risk-aware fuzzy-LTR score while separating ranking from transformation legality. Fourth, it evaluates ranking quality, search cost, sensitivity, inference overhead, an expanded C/C++ benchmark, and a bounded same-benchmark code-LLM context comparison. Repair-success differences are reported but remain secondary because paired binary tests are not statistically conclusive.

The remainder of the paper summarizes related work, presents the method and evaluation protocol, reports the results, discusses validity and reproducibility, and concludes with future work.

2. Background and Related Work

2.1 Pattern- and Learning-Guided Repair

Search- and mutation-based APR systems such as GenProg, RSRepair, and Kali established the feasibility of generate-and-validate repair but also exposed the cost

of large search spaces and the risk of test-suite overfitting Le Goues *et al.* (2012); Qi *et al.* (2014, 2015). Template systems such as PAR, SPR, Nopol, and TBar constrain edits to recurring transformation families and improve transparency, although their efficiency still depends on localization quality, applicability filtering, and pattern order Kim *et al.* (2013); Liu *et al.* (2019); Long and Rinard (2015); Xuan *et al.* (2017). Learning-guided methods such as Prophet, DeepRepair, and Recoder improve prioritization or syntax-aware generation Long and Rinard (2016); White *et al.* (2019); Zhu *et al.* (2021). Fuzzy-AutoRepair acts earlier: it ranks typed repair families before most candidates are instantiated.

2.2 Fuzzy Reasoning and Modern Generative Repair

Fuzzy set theory models gradual membership Zadeh (1965); Mamdani inference expresses expert knowledge through linguistic rules Mamdani and Assilian (1975), while Sugeno-style systems provide compact numerical alternatives Takagi and Sugeno (1985). In this study, fuzzy suitability is advisory: it changes repair-family priority but cannot override hard transformation preconditions.

Neural and pretrained-code-model APR systems, including SequenceR, CoCoNuT, CURE, AlphaRepair, and RepairLLaMA, broaden the generation space Chen *et al.* (2021); Jiang *et al.* (2021); Lutellier *et al.* (2020); Silva *et al.* (2023); Xia and Zhang (2022). Conversational and agentic systems such as ChatRepair, ThinkRepair, and RepairAgent further exploit validation feedback or tool interaction Bouzenia *et al.* (2025); Xia and Zhang (2023); Yin *et al.* (2024). Template–LLM hybrids such as GAMMA, FitRepair, and Template-Guided Program Repair combine explicit structure with model-assisted generation or ingredient selection Huang *et al.* (2025); Xia *et al.* (2023b); Zhang *et al.* (2023). These approaches are complementary to Fuzzy-AutoRepair: their outcomes depend on checkpoint, prompt, context, sampling, tool loop, and budget, whereas the proposed layer produces a reproducible ranking over explicit repair intents. Broader reviews of PLM/LLM-based APR reinforce the need to control these factors before making numerical cross-system claims Xia *et al.* (2023a); Yang *et al.* (2025).

2.3 Research Gap

The gap addressed here occurs after an imperfect localization result is available. Fixed ordering does not represent partial repair evidence; pure learned ordering offers limited rule-level explanation; and neither explicitly separates structural risk from transformation legality. Fuzzy-AutoRepair addresses this gap through pattern-conditioned fuzzy evidence, expert-validated rules, risk-aware fusion, and direct testing of the link between useful-pattern rank and downstream search cost.

3. The Fuzzy-AutoRepair Approach

3.1 Architecture and Region Representation

For each suspicious location s, Fuzzy-AutoRepair receives the bounded AST-level region from SpecPatch Ghanati *et al.* (forthcoming) and encodes it as

$$\mathbf{x}(s) = [x_1, x_2, \ldots, x_{40}], \qquad x_i \in \{0, 1\}. \tag{3.1}$$

Features 1–22 describe fault-local syntax and expression roles; Features 23–40 describe control-flow, data-flow, call, loop, and function context. The vector conditions repair-family suitability at the supplied location; it is not used to claim that a new faulty statement has been localized.

Figure 1 summarizes the processing path. Learned and fuzzy scores are computed independently, fused with a structural-risk term, filtered by hard applicability constraints, and then used to order candidate generation, compilation, failing-test-first execution, regression testing, and trace recording.

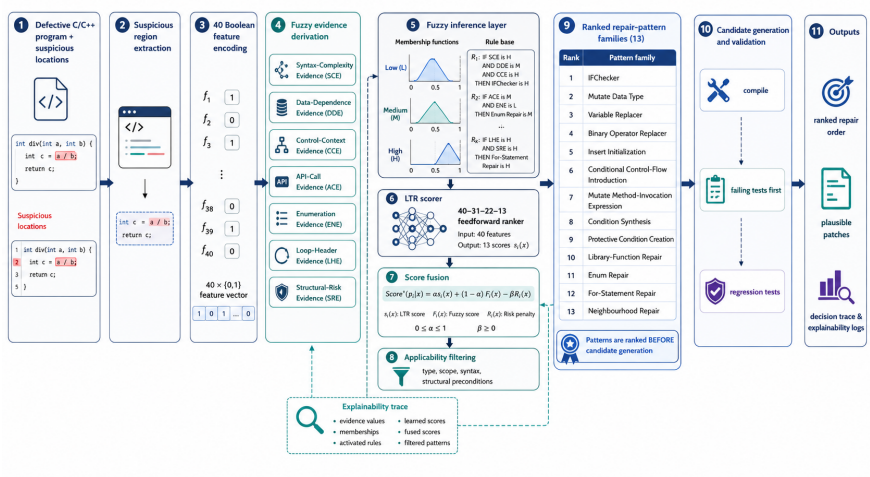

Figure 1: End-to-end Fuzzy-AutoRepair processing pipeline. Learned relevance and fuzzy suitability rank thirteen repair-pattern families before hard applicability checks and AST-level validation.

3.2 Fuzzy Evidence Variables

The 40 Boolean features are grouped into seven evidence variables. For feature group G_k, the crisp evidence value is

$$e_k(\mathbf{x}) = \frac{\sum_{j \in G_k} w_{kj} x_j}{\sum_{j \in G_k} w_{kj}}. \tag{3.2}$$

Each value is mapped to low, medium, and high memberships using the default configuration in Appendix B. Table 1 gives the evidence groups used by the rule base.

Table 1: Fuzzy evidence variables.

Evidence	Main features	Interpretation
Syntax complexity (SCE)	1–10, 19–21	Operators, conditions, literals, return usage, and logical structure.
Data dependence (DDE)	11–16, 22, 29, 37	Variables, arrays, pointers, globals, and nearby definitions.
Control context (CCE)	7–10, 23–27, 32, 36, 39	Branch, loop, label, goto, and terminating control flow.
API call (ACE)	17, 30, 31, 37	Calls, return-constrained calls, void calls, and arguments.
Enumeration (ENE)	18, 33	Resolved enumeration values or accessible members.
Loop header (LHE)	8, 26, 29	For-condition, body, bound, or update dependence.
Structural risk (SRE)	28, 32, 34, 36, 39, 40	Block position, unused variables, termination, labels, goto, and function context.

3.3 Learning-to-Rank Model

Let $\mathbf{y} \in \{0,1\}^{13}$ be the multi-label applicability vector, $R(\mathbf{x})$ the relevant family set, and $\mathcal{P}(\mathbf{x}) = \{(i,j) : i \in R(\mathbf{x}), j \notin R(\mathbf{x})\}$. The selected 40–31–22–13 feed-forward ranker uses ReLU activations, dropout 0.15, and independent sigmoid outputs $z_i(\mathbf{x})$. Its pairwise loss is

$$\mathcal{L}_{\text{rank}} = \frac{1}{|\mathcal{P}(\mathbf{x})|} \sum_{(i,j) \in \mathcal{P}(\mathbf{x})} \log(1 + \exp[-(z_i - z_j)]) \,. \tag{3.3}$$

Training combines this objective with class-weighted binary cross-entropy using $\lambda = 0.5$. Adam uses a learning rate of 0.001, a batch size of 64, a weight decay of 10^{-5}, at most 100 epochs, and early stopping with a patience of 10. The 6,734 development samples are split into 5,388 training and 1,346 validation

samples; 2,244 independent test samples are evaluated only after configuration selection. The selected architecture has 2,274 parameters; a 40–64–32–13 alternative improved validation NDCG@5 by only 0.003 while using 5,133 parameters.

3.4 Rule Base and Fuzzy Suitability

For repair family p_i, the fuzzy layer computes $F_i(\mathbf{x}) \in [0, 1]$. A rule has the form

$$\text{IF } e_a \text{ is } u \text{ AND } e_b \text{ is } v \text{ THEN } F_i \text{ is } q. \tag{3.4}$$

The Mamdani system uses minimum t-norm firing, maximum aggregation, and centroid defuzzification. The initial 46 rules were derived from a documented feature-to-pattern applicability matrix and reviewed independently on 180 development examples by two APR experts and one fuzzy-systems expert. Two adjudication rounds retained 42 rules. Fleiss' $\kappa = 0.81$ for target-family assignment and weighted $\kappa = 0.77$ for ordinal suitability; no held-out test sample was used to tune the rules.

Table 2: Representative rule families.

Family	Representative evidence relation	Main targets
Condition	High SCE and high CCE → strong suitability	IFChecker; condition synthesis/protection
Variable/data	High DDE with medium SCE → strong suitability	Variable replacement; initialization
Operator	High SCE and medium DDE → strong suitability	Binary operator replacement
Invocation	High ACE and low SRE → strong suitability	Invocation; library-function repair
Enum	High ENE → strong suitability	Enum Repair
Loop	High LHE and high CCE → strong suitability	For-Statement Repair
Neighbourhood	High DDE with weak direct evidence → moderate suitability	Neighbourhood Repair
Risk damping	High SRE → reduced suitability	Broad/invasive families

3.5 Fuzzy-LTR Fusion and Hard Applicability

The final order combines learned relevance, fuzzy suitability, and risk:

$$\text{Score}_{\text{fused}}(p_i|\mathbf{x}) = \alpha s_i(\mathbf{x}) + (1-\alpha)F_i(\mathbf{x}) - \beta R_i(\mathbf{x}), \tag{3.5}$$

$$R_i(\mathbf{x}) = \gamma_s \text{SRE}(\mathbf{x}) + \gamma_g \widetilde{G}_i(\mathbf{x}) + \gamma_b B_i, \qquad \gamma_s + \gamma_g + \gamma_b = 1. \tag{3.6}$$

Here $\widetilde{G}_i$ is normalized ingredient-space size and B_i is normalized broad-edit scope. Validation searched $\alpha \in \{0, 0.25, 0.50, 0.75, 1\}$ and $\beta \in \{0, 0.05, 0.10, 0.20\}$; the selected setting is $\alpha = 0.50$, $\beta = 0.10$. Ranking never authorizes an illegal edit: enum, call, loop, type, scope, signature, ingredient, and structural preconditions remain hard filters.

The thirteen ranked families are IFChecker, Mutate Data Type, Variable Replacer, Binary Operator Replacer, Insert Initialization, Conditional Control-Flow Introduction, Mutate Method-Invocation Expression, Condition Synthesis, Protective Condition Creation, Library-Function Repair, Enum Repair, For-Statement Repair, and Neighbourhood Repair. The last four and the underlying candidate-generation substrate are inherited from SpecPatch Ghanati *et al.* (forthcoming); the contribution here is the validated ordering mechanism.

3.6 Candidate Generation and Validation

Applicable families are attempted in decreasing fused-score order. Candidates are instantiated within the budget, duplicates are discarded, compilation is attempted, defect-revealing tests are run before the complete regression suite, and a patch is called plausible only if it compiles and passes all supplied tests. The trace records evidence values, membership degrees, rule activations, learned/fuzzy/risk/fused scores, hard-filter decisions, candidate outcomes, elapsed time, and stopping condition.

Algorithm 3.1. ***Fuzzy-AutoRepair ranking and validation.***

1. *For each suspicious location, compute the 40-feature vector, fuzzy evidence values, memberships, learned scores, fuzzy suitability, risk, and fused score.*

2. *Sort the 13 families and remove those that violate hard syntax, type, scope, signature, or structural constraints.*

3. *Instantiate non-duplicate candidates in ranked order, compile them, run defect-revealing tests, and then run the full regression suite when appropriate.*

4. *Record the decision/validation trace and apply the configured stopping policy.*

4. Experimental Methodology

4.1 Research Questions and Data

The evaluation addresses nine closely related questions grouped into four themes: (RQ1–RQ4) descriptive repair outcomes, selectivity, project sensitivity, and historical runtime context; (RQ5) the contribution of fuzzy-LTR to rank-aware quality and downstream search cost; (RQ6) rule validity and parameter sensitivity; and (RQ7–RQ9) inference overhead, rank–cost mechanism, expanded-benchmark generalization, and a bounded code-LLM context comparison.

The ranking corpus contains 8,978 Code4Bench-derived samples Majd *et al.* (2019a,b), grouped by source problem/submission family. Development uses 5,388 training and 1,346 validation samples; 2,244 samples form the independent test partition. The end-to-end study uses the original 69 C/C++ defects and BUGSC++ An *et al.* (2023). All 209 available BUGSC++ versions were attempted; 181 met three predeclared criteria: successful build, an executable test harness with at least one reproducible failing test, and an available suspicious location under the common protocol. Paired LTR-only and hybrid runs use identical locations, repair families, hard constraints, budgets, compiler, tests, stopping policy, and environment.

A Qwen2.5-Coder-7B-Instruct context baseline Hui *et al.* (2024); Qwen Team (2024) is run on the original 69 defects using the same suspicious statements and tests and a budget of ten generated candidates per defect. Because its generative repair space and GPU execution differ from the pattern-based system, it is used for context rather than a causal superiority claim.

4.2 Outcomes and Statistical Analysis

A plausible patch compiles and passes all defect-revealing and regression tests. A correct-first patch is the first plausible patch manually judged consistent with the intended developer change. These outcomes are kept separate because test-suite adequacy does not establish semantic equivalence.

Binary paired outcomes use exact McNemar tests. Candidate counts, compilation attempts, and runtime use paired Wilcoxon signed-rank tests with rank-biserial effect sizes. Proportions receive two-sided 95% confidence intervals. Sensitivity/architecture results are summarized over five fixed seeds, and rank–cost associations use two-sided Spearman correlation with 10,000-sample bootstrap confidence intervals.

For sample n, let $\pi_{n,1:k}$ be the first k ranked families and R_n the relevant set. The principal ranking measures include

$$\text{Hit@}k(n) = \mathbf{1}[R_n \cap \pi_{n,1:k} \neq \varnothing], \tag{4.7}$$

$$\text{MacroRecall@}k = \frac{1}{N}\sum_{n=1}^{N}\frac{|R_n \cap \pi_{n,1:k}|}{|R_n|}. \tag{4.8}$$

MRR, MAP, and NDCG are used as complementary rank-aware metrics Burges *et al.* (2005); Burges (2010); Järvelin and Kekäläinen (2002). Internal fuzzy-LTR versus LTR-only comparisons support causal claims; historical APR comparisons and the code-LLM run are interpreted descriptively unless the repair space and execution protocol are equivalent.

5. Results

5.1 Ranking Quality and End-to-End Search Cost

On the independent 2,244-sample test set, hybrid fuzzy-LTR improves Hit@1 from 0.45 to 0.50, macro Recall@5 from 0.806 to 0.846, and NDCG@5 from 0.78 to 0.84 relative to LTR-only ordering. Table 3 shows the complete ranking comparison.

Table 3: Rank-aware performance on the independent test set.

Ordering	Hit@1	Hit@3	Recall@5	MRR	NDCG@5
Random	0.18	0.43	0.386	0.31	0.46
Fixed	0.31	0.58	0.611	0.46	0.62
Fuzzy only	0.42	0.70	0.781	0.58	0.75
LTR only	0.45	0.74	0.806	0.61	0.78
Hybrid fuzzy-LTR	**0.50**	**0.80**	**0.846**	**0.67**	**0.84**

Under the identical 69-defect repair protocol, hybrid fuzzy-LTR reduces generated candidates from 914 to 620 per defect (32.2%), compilation attempts from 337 to 238 (29.4%), and mean runtime from 49.38 to 41.70 minutes (15.6%). Plausible repairs increase from 33/69 to 36/69 and correct-first repairs from 14/69 to 17/69, but these binary changes remain secondary. Table 4 summarizes the ablation.

Exact McNemar tests give $p = 0.375$ for both plausible and correct-first success (discordant pairs 1 versus 4), so the study does not claim improved repair effectiveness. In contrast, Wilcoxon tests support reductions in candidates and compilations at $p < 0.001$ and runtime at $p = 0.006$, with rank-biserial effect sizes of 0.72, 0.66, and 0.46. Median reductions are 248 candidates, 79 compilation attempts, and 6.10 minutes.

Table 4: End-to-end ablation on the same 69 defects.

Ordering	Plaus.	Correct	Cand.	Comp.	Time
Random	30/69	8/69	1860	702	76.80
Fixed	32/69	10/69	1435	534	63.20
LTR only	33/69	14/69	914	337	49.38
Fuzzy only	34/69	15/69	780	294	45.90
Hybrid	36/69	17/69	**620**	**238**	**41.70**

5.2 Descriptive Context, Sensitivity, and Rule Validation

The baseline repairs 33/69 defects plausibly (47.8%; 95% CI 36.5–59.4) and produces 14/69 correct-first patches (20.3%; 95% CI 12.5–31.2). Historical counts for Prophet, SPR, Kali, and GenProg are reported only as context because those systems were not rerun in the same environment. PHP contributes 10 of the 14 baseline correct-first outcomes, so cross-project conclusions remain conservative. The historical AutoRepair–SPR project-level runtime difference is also non-significant despite a lower arithmetic mean (49.38 versus 69.41 minutes).

Validation selects $\alpha = 0.50$ and $\beta = 0.10$. Across the α sweep, Hit@1 ranges from 0.42 to 0.50, and the balanced configuration gives the lowest observed candidate count (620) and runtime (41.70 minutes). Reasonable triangular, trapezoidal, Gaussian, and breakpoint-shifted membership alternatives change NDCG@5 by at most 0.02 and candidate count by 7.9%. At $\alpha = 0.50$, increasing β from 0 to 0.10 lowers candidates from 654 to 620 without reducing NDCG@5; $\beta = 0.20$ lowers candidates to 608 but reduces NDCG@5 to 0.82.

The selected 40–31–22–13 network obtains test micro precision/recall/F1 of 0.843/0.817/0.830 and macro precision/recall/F1 of 0.795/0.751/0.768. The three-expert process retains 42 of 46 rules, with Fleiss' $\kappa = 0.81$, weighted $\kappa = 0.77$, and 98.1% held-out rule coverage. Leave-one-rule-family-out analysis reduces NDCG@5 by 0.008–0.031, with the largest losses for condition, invocation, and loop rule families.

5.3 Overhead, Rank–Cost Mechanism, and Generalization

Fuzzy processing averages 1.31 ms per suspicious region (95th percentile 1.92 ms), 0.41 s per defect, and 4.2 MB additional peak memory, approximately 0.016% of total repair runtime. The rank of the first plausible-producing pattern correlates with generated candidates ($\rho = 0.74$, 95% bootstrap CI 0.61–0.83, $p < 0.001$); candidate count correlates with compilations ($\rho = 0.91$) and runtime ($\rho = 0.82$), both $p < 0.001$. Among 44 defects whose first useful pattern moves upward, the

median reduction is 271 candidates and 6.7 minutes. This supports the intended mechanism: better family ordering reaches useful candidates earlier and avoids unnecessary validation.

On the 181 eligible BUGSC++ defects, hybrid fuzzy-LTR improves Hit@1 from 0.43 to 0.49 and NDCG@5 from 0.76 to 0.82. Candidates decrease from 1,128 to 781 (30.8%), compilations from 418 to 300 (28.2%), and runtime from 56.8 to 48.3 minutes (15.0%); all three paired cost endpoints have $p < 0.001$. Plausible repairs change from 68/181 to 74/181 and correct-first repairs from 27/181 to 31/181, but exact McNemar tests remain non-significant.

For Qwen2.5-Coder-7B-Instruct, the same 69-defect context run yields 35 plausible and 15 correct-first repairs under ten samples per defect, compared with 36 and 17 for hybrid fuzzy-LTR. Compilation attempts are 287 versus 238, and reported wall-clock times are 52.6 versus 41.7 minutes. Because the repair spaces and hardware differ, these values are contextual and do not support a general superiority claim.

6. Discussion

Fuzzy-AutoRepair addresses a narrow but consequential decision problem: a suspicious statement can simultaneously provide partial evidence for condition, variable, invocation, loop, or neighbourhood repair. The fuzzy layer makes this evidence explicit, the LTR component contributes learned relevance, the risk term discourages broad edits, and hard applicability checks preserve transformation legality. The resulting trace can explain which evidence and rules affected priority without treating fuzzy suitability as permission to perform an unsafe transformation.

The strongest empirical result is mechanism-oriented. Ranking quality improves before candidate generation, useful-pattern rank is strongly associated with candidate cost, the paired candidate/compilation/runtime reductions are statistically supported, and inference overhead is negligible. Sensitivity tests further indicate that the gain is not tied to one narrow membership or fusion setting. In contrast, plausible and correct-first counts are deliberately not promoted to primary effectiveness claims because their paired differences are statistically inconclusive.

Modern LLM-based APR offers a broader generation space but introduces model-, prompt-, context-, sampling-, tool-, and hardware-dependent behavior. A practical integration path is therefore to use fuzzy-LTR as an intent-selection or constraint layer that supplies a small ranked set of typed repair intents and hard applicability conditions to a generative model. Such a hybrid requires a same-protocol evaluation and is left to future work.

7. Threats to Validity

Construct validity. Passing a finite test suite establishes plausibility rather than semantic equivalence. Correct-first judgments reduce this limitation but still depend on the assessment protocol and augmented tests. Fuzzy suitability is meaningful only insofar as it predicts earlier relevant-family discovery, lower search cost, or clearer decision provenance.

Conclusion validity. Historical APR comparisons are not paired by environment and are therefore descriptive. The principal controlled evidence is LTR-only versus hybrid fuzzy-LTR on identical defects, locations, budgets, compiler, tests, and stopping policy. Binary repair-success differences are non-significant under exact McNemar testing, while paired cost endpoints support the search-efficiency claim.

External validity. The original 69 defects are supplemented by 181 eligible BUGSC++ defects selected from all 209 attempted versions using predeclared build/test/location criteria An *et al.* (2023). This broadens the evaluation but does not cover all multi-file, concurrency, security, weak-test, or repair-space scenarios.

Reproducibility validity. The manuscript reports the complete feature schema, fuzzy evidence construction, membership configuration, representative rule logic, rule-validation protocol, ranker architecture/training settings, fusion equations and search ranges, hard applicability boundary, benchmark-selection criteria, metrics, and statistical procedures. Exact bit-for-bit replication would additionally require the complete SpecPatch-shared source tree, trained checkpoints, processed split files, local benchmark snapshots, internal rule implementation, raw candidate traces, and build/test logs; these implementation assets are not distributed with this manuscript. This boundary distinguishes methodological reimplementation from exact execution replication.

8. Reproducibility and Implementation Availability

Code4Bench Majd *et al.* (2019a,b) and BUGSC++ An *et al.* (2023) are public third-party benchmark resources and should be obtained from their original distributions. The study reports the study-specific split sizes, BUGSC++ eligibility criteria, attempted/eligible counts, repair budgets, stopping policy, compilation/test protocol, and statistical analyses needed to reconstruct the evaluation design.

The complete Fuzzy-AutoRepair source tree and SpecPatch-shared execution infrastructure are not publicly distributed with this manuscript. The underlying

suspicious-region representation, thirteen-pattern catalogue, AST-level candidate-generation substrate, and compilation/test pipeline belong to SpecPatch Ghanati *et al.* (forthcoming); the present paper contributes and specifies the fuzzy-LTR ordering mechanism over that substrate. Accordingly, the reproducibility claim is limited to methodological transparency and independent reimplementation, not bit-for-bit replay from a public archive.

9. Conclusion

Fuzzy-AutoRepair adds an expert-validated, interpretable pre-generation ordering layer to the SpecPatch C/C++ repair pipeline. It combines learned relevance, fuzzy suitability, structural-risk damping, and hard applicability constraints without changing raw fault-localization scores or introducing a new candidate generator.

Relative to LTR-only ordering, the hybrid improves Hit@1 from 0.45 to 0.50, macro Recall@5 from 0.806 to 0.846, and NDCG@5 from 0.78 to 0.84 on 2,244 independent ranking samples. On 69 defects, it reduces generated candidates, compilation attempts, and runtime by 32.2%, 29.4%, and 15.6%; on 181 eligible BUGSC++ defects, the corresponding reductions are 30.8%, 28.2%, and 15.0%. Paired cost reductions are statistically supported, while repair-success differences remain inconclusive.

Future work will evaluate broader C/C++ scenarios, including multi-file, concurrency, security, and weak-test defects, and will study fuzzy-ranked repair intents as retrieval or control signals for LLM-based repair under matched budgets and validation protocols. Larger paired benchmarks and independent reimplementations will further test the portability of the observed ranking–cost mechanism.

References

An G., Kwon M., Choi K., Yi J. and Yoo S. (2023), BUGSC++: A highly usable real world defect benchmark for C/C++, *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, 2034–2037.

Bouzenia I., Devanbu P. and Pradel M. (2025), RepairAgent: An autonomous, LLM-based agent for program repair, *Proceedings of ICSE 2025*.

Burges C., Shaked T., Renshaw E., Lazier A., Deeds M., Hamilton N. and Hullender G. (2005), Learning to rank using gradient descent, *Proceedings of ICML 2005*, 89–96.

Burges C. J. C. (2010), From RankNet to LambdaRank to LambdaMART: An overview, *Microsoft Research Technical Report MSR-TR-2010-82*.

Chen Z., Kommrusch S., Tufano M., Pouchet L., Poshyvanyk D. and Monperrus M. (2021), SequenceR: Sequence-to-sequence learning for end-to-end program repair, *IEEE Transactions on Software Engineering,* **47**(9), 1940–1956.

Gazzola L., Micucci D. and Mariani L. (2019), Automatic software repair: A survey, *IEEE Transactions on Software Engineering,* **45**(1), 34–67.

Ghanati A., Parsa S. and Izadkhah H. (2025), SpecPatch: Specification-aware fault localization and constraint-guided program repair for C/C++ systems, *International Journal of Mathematical Modelling & Computations*. in press.

Huang K., Zhang J., Meng X. and Liu Y. (2025), Template-guided program repair in the era of large language models, *Proceedings of ICSE 2025*.

Hui B. *et al.* (2024), Qwen2.5-Coder technical report, *arXiv:2409.12186*.

Järvelin K. and Kekäläinen J. (2002), Cumulated gain-based evaluation of IR techniques, *ACM Transactions on Information Systems,* **20**(4), 422–446.

Jiang N., Lutellier T. and Tan L. (2021), CURE: Code-aware neural machine translation for automatic program repair, *Proceedings of ICSE 2021*, 1161–1173.

Kim D., Nam J., Song J. and Kim S. (2013), Automatic patch generation learned from human-written patches, *Proceedings of ICSE 2013*, 802–811.

Le Goues C., Nguyen T., Forrest S. and Weimer W. (2012), GenProg: A generic method for automatic software repair, *IEEE Transactions on Software Engineering,* **38**(1), 54–72.

Liu K., Koyuncu A., Kim D. and Bissyandé T. F. (2019), TBar: Revisiting template-based automated program repair, *Proceedings of ISSTA 2019*, 31–42.

Long F. and Rinard M. (2015), Staged program repair with condition synthesis, *Proceedings of ESEC/FSE 2015*, 166–178.

Long F. and Rinard M. (2016), Automatic patch generation by learning correct code, *Proceedings of PLDI 2016*, 298–312.

Lutellier T., Pham H., Pang L., Li Y., Wei M. and Tan L. (2020), CoCoNuT: Combining context-aware neural translation models using an ensemble for program repair, *Proceedings of ISSTA 2020*, 101–114.

Majd A., Vahidi-Asl M., Khalilian A., Baraani-Dastjerdi A. and Zamani B. (2019a), Code4Bench: A multidimensional benchmark of Codeforces data for different program analysis techniques, *Journal of Computer Languages,* **53**, 38–52.

Majd A., Vahidi-Asl M., Khalilian A., Baraani-Dastjerdi A. and Zamani B. (2019b), Code4Bench: A multidimensional benchmark of Codeforces data for different program analysis techniques, Zenodo, version 1.0.0.

Mamdani E. H. and Assilian S. (1975), An experiment in linguistic synthesis with a fuzzy logic controller, *International Journal of Man-Machine Studies,* **7**(1), 1–13.

Qi Y., Mao X., Lei Y., Dai Z. and Wang C. (2014), The strength of random search on automated program repair, *Proceedings of ICSE 2014*, 254–265.

Qi Z., Long F., Achour S. and Rinard M. (2015), An analysis of patch plausibility and correctness for generate-and-validate patch generation systems, *Proceedings of ISSTA 2015*, 24–36.

Qwen Team (2024), Qwen2.5-Coder-7B-Instruct, Hugging Face model card.

Silva A., Fang S. and Monperrus M. (2023), RepairLLaMA: Efficient representations and fine-tuned adapters for program repair, *arXiv:2312.15698*.

Takagi T. and Sugeno M. (1985), Fuzzy identification of systems and its applications to modeling and control, *IEEE Transactions on Systems, Man, and Cybernetics,* **SMC-15**(1), 116–132.

White M., Tufano M., Martinez M., Monperrus M. and Poshyvanyk D. (2019), Sorting and transforming program repair ingredients via deep learning code similarities, *Proceedings of SANER 2019*, 479–490.

Xia C. S. and Zhang L. (2022), Less training, more repairing please: Revisiting automated program repair via zero-shot learning, *Proceedings of ESEC/FSE 2022*.

Xia C. S. and Zhang L. (2023), Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT, *arXiv:2304.00385*.

Xia C. S., Wei Y. and Zhang L. (2023a), Automated program repair in the era of large pre-trained language models, *Proceedings of ICSE 2023*.

Xia C. S., Ding Y. and Zhang L. (2023b), The plastic surgery hypothesis in the era of large language models, *Proceedings of ASE 2023*.

Xuan J. *et al.* (2017), Nopol: Automatic repair of conditional statement bugs in Java programs, *IEEE Transactions on Software Engineering,* **43**(1), 34–55.

Yang B., Cai Z., Liu F., Le B., Zhang L., Bissyandé T. F., Liu Y. and Tian H. (2025), A survey of LLM-based automated program repair: Taxonomies, design paradigms, and applications, *arXiv:2506.23749*.

Yin X., Ni C., Wang S., Li Z., Zeng L. and Yang X. (2024), ThinkRepair: Self-directed automated program repair, *Proceedings of ISSTA 2024*.

Zadeh L. A. (1965), Fuzzy sets, *Information and Control,* **8**(3), 338–353.

Zhang Q., Fang C., Zhang T., Yu B., Sun W. and Chen Z. (2023), GAMMA: Revisiting template-based automated program repair via mask prediction, *Proceedings of ASE 2023*, 535–547.

Zhu Q., Sun Z., Xiao Y., Zhang W., Yuan K., Xiong Y. and Zhang L. (2021), A syntax-guided edit decoder for neural program repair, *Proceedings of ESEC/FSE 2021*, 341–353.

A. Complete Feature Definitions

Table A1: Complete forty-feature Boolean schema used for each suspicious C/C++ region.

#	Fault-local/contextual feature	#	Fault-local/contextual feature
1	Is the code a logical expression?	2	Is the code an assignment expression?
3	Is an assignment used as a logical expression?	4	Does the faulty expression contain a relational operator?
5	Does the faulty expression contain a logical operator?	6	Does the faulty expression contain an arithmetic operator?
7	Is the expression part of an if condition?	8	Is the expression part of a for condition?
9	Is the expression part of a while condition?	10	Is the expression part of a do-while condition?
11	Does the expression contain a pointer variable?	12	Does the expression use a non-pointer variable?
13	Does the expression use a struct or struct field?	14	Does the expression use an array or array element?
15	Does an array access use a constant numerical index?	16	Does an array access use a variable index?
17	Does the expression contain a function call?	18	Does the expression contain a named enumeration value?
19	Does the expression contain a numeric literal?	20	Does the expression contain a Boolean keyword or Boolean variable?
21	Is the expression contained in a return statement?	22	Does the expression use a global variable?
23	Is the expression in the else branch of an if statement?	24	Is the expression in the then branch of an if statement?
25	Is the expression inside a while-loop body?	26	Is the expression inside a for-loop body?
27	Is the expression inside a do-while body?	28	Does the enclosing block contain an unused variable?

#	Fault-local/contextual feature	#	Fault-local/contextual feature
29	Is a read value defined or updated in the enclosing block or a directly data-dependent predecessor?	30	Does the enclosing block contain a value-returning call whose result has a constrained range?
31	Does the enclosing block contain a void-returning function call?	32	Does the basic block terminate the function's control flow?
33	Is the referenced member or expression publicly accessible?	34	Is the expression at the beginning of a block?
35	Is the expression within the condition of a ternary operator?	36	Does the enclosing block contain a goto statement?
37	Does the block contain a call whose input parameter is used by the faulty expression?	38	Is the expression inside a function with no input parameters?
39	Does the enclosing block contain a label?	40	Is the expression inside a function with at least one input parameter?

B. Default Fuzzy Membership Configuration

Each evidence value e_k is normalized to $[0, 1]$. Low membership is trapezoidal with full membership in $[0, 0.20]$ and zero at 0.45; medium is triangular with peak 0.50 and zero at 0.20 and 0.80; high is trapezoidal with zero at 0.55 and full membership in $[0.80, 1]$.

Table B1: Default membership-function supports and cores.

Term	Support/core	Interpretation
Low	support $[0.00, 0.45]$, core $[0.00, 0.20]$	Weak or absent evidence.
Medium	support $[0.20, 0.80]$, peak 0.50	Partial evidence that should not dominate ranking alone.
High	support $[0.55, 1.00]$, core $[0.80, 1.00]$	Strong evidence that raises priority when hard preconditions hold.