

Graph-Evidence Repository Repair: Risk-Calibrated Retrieval and Patch Gating on SWE-Bench

Haitao Gong

GAM Trading Ltd., Saskatoon, SK S7K 6N8, Canada

Abstract: Repository-scale issue repair requires more than patch generation: it requires evidence retrieval, test-grounded rejection, uncertainty calibration, and an explicit decision about whether a candidate deserves scarce verification capacity. This paper proposes RAGE-Repair, a graph-evidence repair method evaluated with the public SWE-bench Lite corpus. The case study loads 300 SWE-bench Lite instances from 12 repositories and analyzes issue length, patch size, test-patch effort, and changed-file scope. The method constructs a repository evidence lattice, estimates residual ambiguity, penalizes tail patch risk with CVaR, and admits a candidate only when expected repair value exceeds its test and review price. The median problem statement contains 1044 characters, the median gold patch has 20 lines, and the top repository contributes 114 instances. Native derivations establish the robust risk bound, primal-dual capacity controller, online regret guarantee, and graph-specific patch gate. The resulting design makes each accepted or rejected repair traceable to evidence support, uncertainty, and verification cost.

Keywords: software engineering; automated program repair; SWE-bench; risk calibration; repository-level agents

1. Repository Repair Problem and Research Questions

Repository-scale repair is an evidence-integration problem before it is a code-generation problem. A useful repair system must reconcile issue intent, failing tests, dependency structure, and candidate-patch consequences while preserving a trace that a maintainer can audit [1–9]. This paper begins from that repair chain and asks where uncertainty should enter before a patch is allowed to consume build and review capacity.

The target problem is repository-scale repair systems must convert natural-language issues, tests, and repository context into safe patch actions. This problem is representative of daily software-engineering work because the decision surface is mixed: code structure, natural-language reports, tests, process history, and human capacity all matter. A purely predictive model can assign a score, but a software team needs more than a score. It needs calibrated uncertainty, an action threshold, a budget mechanism, and a record of why a decision was taken.

This paper proposes RAGE-Repair, whose central contribution is a CVaR-corrected retrieval-and-gating method that treats patch generation as auditable software-engineering risk control. The method is deliberately conservative. It does not claim to replace developers, reviewers, testers, or security engineers. Instead, it exposes risk terms that can be monitored, tuned, capped, and discussed during engineering review.

The empirical basis is the public SWE-bench Lite dataset [10].

A recurring challenge is nonstationarity. Releases, dependency upgrades, framework migrations, new contributors, and production incidents can change the distribution of software events. The proposed method therefore uses a rolling residual distribution and a tail-risk correction rather than a fixed threshold. The same idea has appeared in risk optimization and robust decision-making, but here it is adapted to everyday software-engineering decisions [11,12].

The paper also treats cross-domain validation as a useful methodological analogy. Mature decision systems separate model construction, calibration, resource constraints, and independent empirical validation. The related-work synthesis makes those connections explicitly instead of using a single undifferentiated citation block.

The case-study results are intentionally modest and diagnostic. The median problem statement contains 1044 characters, the median gold patch has 20 lines, and the top repository contributes 114 instances. These statistics do not by themselves prove deployment value, but they show that the dataset contains enough heterogeneity to stress an adaptive method. The figures and tables therefore focus on distributions, correlations, tails, and concentration rather than only on a single average.

The remainder of the paper follows a task-specific research sequence. Section 2 positions retrieval-augmented repair, Section 3 constructs the repair evidence graph, Sections 4 and 5 develop the inference and robust patch gate, and Sections 6–8 move from SWE-bench evidence to failure topology and deployment limits. Section 9 closes the argument.

2. Retrieval-Augmented Repair and the Evidence Gap

Classical software engineering emphasizes modularity, requirements clarity, verification, maintainability, and economic trade-offs [1–9].

Mining software repositories established that commits, issues, tests, code metrics, and social process signals can predict defect and maintenance outcomes [13–19].

Research on automated repair, debugging, fault localization, code review, and continuous integration provides operational context [20–30].

Distributed systems and large-scale infrastructure add another dimension. Ordering, coordination, consistency, and batch computation shape the way software systems are built and observed [31–35].

Independent model validation provides a direct analogy for repository-scale evaluation. Field bioretention studies distinguish process-model development from validation against observed operating data [36–39].

Instrumentation research offers a second analogy: sensitivity is created by an engineered observation path, not by a classifier alone. Plasmonic antenna and frequency-tunable sensing studies combine physical structure with measurable resonance behavior [40–43].

Resource-aware learning is especially relevant to deployable software automation. Energy-efficient edge reconstruction demonstrates that latency and computation belong inside the method definition, dynamic dropout treats training effort as an adaptive quantity [44–46].

Finally, recent multimodal and multi-task systems show that evidence fusion should preserve modality-specific uncertainty paper’s graph, hierarchical, interaction, or hazard representation without implying that domain-specific results transfer unchanged [47–54].

The research gap is therefore not the absence of predictors. The gap is the absence of a compact, auditable, risk-calibrated decision framework that turns heterogeneous repository evidence into budget-aware engineering actions. The proposed method addresses this gap by combining residual calibration, CVaR tail risk, action utility, and dual-price budget control.

3. Repair Evidence Graph and Observable State

The system is observed at discrete engineering decision points. A decision point may be an issue triage event, a failing test, a pull-request update, a vulnerability disclosure, or a release-candidate gate. The index t denotes the current decision point and T denotes the evaluation horizon.

The variable x_t denotes the feature representation available before action. It may include source-code measurements, textual descriptions, test results, dependency metadata, repository ownership, and process

signals. The target y_t denotes the observed outcome after the action, while $\hat{y}_t$ is the model prediction available before the outcome is known.

The policy π maps observable evidence into an engineering action. Actions may include scheduling a repair attempt, ranking a suspicious file, publishing a review comment, escalating a vulnerability, or abstaining when uncertainty is too high. The action has value, cost, risk, and human-interruption components.

The notation is chosen to keep the paper auditable. The loss L_t measures decision error or operational harm. The residual e_t measures the difference between observed and predicted outcomes. The ambiguity radius ρ_t summarizes how unstable recent residuals are. The budget price v_t summarizes scarce engineering capacity (see Table 1).

Table 1. Main notation used by the proposed software-engineering method.

Symbol	Meaning	Operational Interpretation
t, T	decision index and horizon	sequence of software-engineering events
x_t	feature vector	code, text, test, and process evidence
$y_t, \hat{y}_t$	outcome and prediction	observed and predicted engineering result
e_t	prediction residual	calibration signal
ρ_t	ambiguity radius	tail uncertainty of recent residuals
L_t	loss	defect, review, repair, or vulnerability risk
a_t	action	triage, repair, review, patch, or abstention decision
v_t	capacity price	scarcity of engineering budget
$CVaR_\alpha$	tail-risk measure	expected loss beyond a high quantile

4. RAGE-Repair Inference Pipeline

The proposed method, RAGE-Repair, is built as a control layer rather than as a single classifier. It receives repository evidence, estimates risk, corrects that risk for recent residual tails, and then decides whether the action is worth the capacity it consumes.

The method has three design principles. First, every symbol is tied to an observable artifact. Second, the tail correction is explicit and can be disabled or capped in deployment. Third, the budget price is logged so that teams can see when automation is acting conservatively because human or compute capacity is scarce.

The algorithm table (see Table 2) is written as a deployment interface. Each line corresponds to an engineering step that can be tested independently: parsing, scoring, calibration, budget control, action selection, and audit logging.

Table 2. RAGE-Repair algorithmic procedure.

Line	Operation	Audit signal
1	Parse issue text, repository metadata, tests, and historical residuals.	Feature map x_t
2	Retrieve similar resolved issues and patch files.	Evidence neighborhood
3	Compute CVaR-corrected repair priority score.	Tail risk
4	Generate candidate patch plan and test plan.	Action set
5	Apply budget-aware acceptance gate.	Engineering-capacity price
6	Log residual, uncertainty radius, and final decision.	Incident-review record

5. Robust Patch-Gating Derivation

The derivation proceeds from the decision objective to calibration, tail risk, budgeted action selection, regret, and cost-normalized improvement. The equations are included as Word native equations with numbering so that the method can be inspected and revised like a conventional software-engineering model.

5.1. Repair Loss and Observable Repository Evidence

The derivation begins with the operational decision itself. Each random quantity is tied to an observable artifact, and every penalty is tied to measurable human, compute, or release effort.

Let t index a repository-scale issue repair decision, let T be the number of decisions in the study horizon, and let π denote the policy that maps evidence to an action. Before Equation (1), y_t is defined as the observed outcome, $\hat{y}_t$ as its prediction, ℓ_t as prediction loss, C_t as operational cost, and A_t as action churn. The nonnegative coefficients λ_c and λ_a convert these terms to a common risk scale. Linearity of expectation then produces the cumulative policy objective.

$$\mathcal{R}(\pi) = \mathbb{E} \sum_{t=1}^T \left[\ell(y_t, \hat{y}_t) + \lambda_c C_t + \lambda_a A_t \right] \quad (1)$$

To expose what the policy observes, r_t denotes repository state, d_t denotes textual or dependency evidence, and m_t denotes process metadata. The feature operator ϕ maps these artifacts to x_t , while f_θ is the model parameterized by θ . Equation (2) defines the prediction $\hat{y}_t$ and residual e_t ; this residual is the measurable quantity used in every later calibration step.

$$x_t = \phi(r_t, d_t, m_t), \quad \hat{y}_t = f_\theta(x_t), \quad e_t = y_t - \hat{y}_t \quad (2)$$

5.2. Retrieval Residuals and Repository-Shift Ambiguity

The next step replaces a stationary noise assumption with a rolling empirical law. A Wasserstein neighborhood represents sampling error and local process drift, while duality converts the uncertainty set into a finite audit-record objective.

Let N_t be the number of recent residuals retained at decision t , and let δ_{e_i} denote a unit point mass at residual e_i . Equation (3) is the empirical residual law obtained by assigning equal probability $1/N_t$ to each retained observation. This construction converts a sequence of audit records into a probability object without assuming Gaussian errors.

$$\hat{\mathbb{P}}_t = \frac{1}{N_t} \sum_{i=t-N_t+1}^t \delta_{e_i} \quad (3)$$

The symbol W_1 denotes the order-one Wasserstein distance between probability laws, and ρ_t is a nonnegative ambiguity radius. Equation (4) defines $\mathcal{P}_t(\rho_t)$ as every distribution lying within ρ_t of the empirical law. The true next-event law is therefore allowed to move locally instead of being equated to the historical sample.

$$\mathcal{P}_t(\rho_t) = \left\{ \mathbb{P} : W_1(\mathbb{P}, \hat{\mathbb{P}}_t) \leq \rho_t \right\} \quad (4)$$

Let $L_\theta(e)$ be the loss induced by residual e under model parameters θ . Equation (5) defines the distributionally robust risk as the worst expected loss over the ambiguity set. The supremum is the formal point at which distribution shift enters the method: a larger ρ_t forces the controller to protect against a broader family of plausible residual laws.

$$\mathcal{R}_t^{rob}(\theta) = \sup_{\mathbb{P} \in \mathcal{P}_t(\rho_t)} \mathbb{E}_{e \sim \mathbb{P}} [L_\theta(e)] \quad (5)$$

To derive a computable objective, introduce λ as the nonnegative multiplier assigned to the Wasserstein constraint. Kantorovich duality transforms the distributional supremum into Equation (6): a one-dimensional infimum over λ plus N_t independent pointwise suprema. This step explains why robust calibration can be implemented without enumerating probability distributions.

$$\mathcal{R}_t^{rob}(\theta) = \inf_{\lambda \geq 0} \left\{ \lambda \rho_t + \frac{1}{N_t} \sum_{i=1}^{N_t} \sup_e [L_\theta(e) - \lambda |e - e_i|] \right\} \quad (6)$$

Assume L_θ is K_θ -Lipschitz in the residual argument, meaning that its change is bounded by K_θ times the residual displacement. For any λ at least K_θ , the inner supremum in Equation (6) is attained at e_i . Substitution yields Equation (7), which separates empirical risk from the explicit robustness premium $K_\theta \rho_t$.

$$\mathcal{R}_t^{rob}(\theta) \leq \frac{1}{N_t} \sum_{i=1}^{N_t} L_\theta(e_i) + K_\theta \rho_t \quad (7)$$

Consequently, the robust penalty is linear in the ambiguity radius whenever the loss is Lipschitz. This is the

key simplification that makes the uncertainty correction transparent and computationally tractable.

The constants C_x and κ in Equation (8) control sampling uncertainty and tail sensitivity, respectively; δ is a user-selected failure probability and α is the CVaR confidence level. The first radius term decreases as N_t grows, whereas the second expands when recent absolute residuals have a heavy upper tail. Thus the radius responds to both sample scarcity and observed instability.

$$\rho_t = C_x \sqrt{(\log(2/\delta)/N_t) + \kappa \text{CVaR}_\alpha(e)} \quad (8)$$

5.3. Tail Loss of Candidate Patches

Average error can conceal the rare cases that dominate practical cost. The Rockafellar-Uryasev representation is therefore developed in population, sample, and differentiated forms so that the source of every tail update is visible.

Let L be a nonnegative random loss and let ζ be a candidate upper-quantile threshold. The positive-part operator $(u)_+ = \max(u, 0)$. Equation (9) defines conditional value at risk at level α by optimizing ζ ; at the optimum, ζ is a value-at-risk threshold and the second term is the mean excess loss beyond that threshold.

$$\text{CVaR}_\alpha(L) = \min_{\zeta} \left[\zeta + \frac{1}{1-\alpha} \mathbb{E}(L - \zeta)_+ \right] \quad (9)$$

For the retained losses L_1 through L_{N_t} , replace the expectation in Equation (9) with its empirical average. Equation (10) is the resulting finite-sample convex program. It can be solved by sorting the losses or by joint gradient optimization, so the tail correction remains reproducible from the logged residual window.

$$\widehat{\text{CVaR}}_\alpha(L) = \min_{\zeta} \left[\zeta + \frac{1}{N_t(1-\alpha)} \sum_{i=1}^{N_t} (L_i - \zeta)_+ \right] \quad (10)$$

Let ζ^* denote the minimizer of Equation (10), and assume no loss lies exactly at that threshold. Differentiating the positive-part terms gives Equation (11). Only tail observations with L_i greater than ζ^* contribute to the CVaR gradient, and their gradients are amplified by $1/[N_t(1-\alpha)]$. This establishes the claimed tail emphasis algebraically.

$$\nabla_{\theta} \widehat{\text{CVaR}}_\alpha = \frac{1}{N_t(1-\alpha)} \sum_{i=1}^{N_t} 1\{L_i > \zeta^*\} \nabla_{\theta} L_i \quad (11)$$

This derivative also supplies an implementation test: examples below the optimized quantile must contribute zero to the tail gradient, while examples above it must receive the same normalization factor.

The symbol s_t denotes the operational priority score, β in $[0, 1)$ is a smoothing factor, $g_{\theta}(x_t)$ is the model evidence score, and η scales the empirical CVaR correction. Equation (12) follows from exponential smoothing: the previous score receives weight β and current robust evidence receives weight $1-\beta$.

$$s_t = \beta s_{t-1} + (1-\beta) \left[g_{\theta}(x_t) + \eta \widehat{\text{CVaR}}_\alpha(L_t) \right] \quad (12)$$

5.4. Test and Review Capacity Constraints

The target action competes for finite operational capacity. The constrained primal problem is converted to a Lagrangian, its KKT conditions are stated, and an online dual update is obtained for unknown future workload.

Let a_t be the action chosen at decision t , $L_t(a_t)$ its expected engineering loss, and $B_t(a_t)$ the consumed review, test, or remediation capacity. The constant B^* is the allowed average capacity per decision. Equation (13) states the finite-horizon constrained problem whose budget is $T B^*$; this is the optimization problem that the later threshold rule approximates online.

$$\min_{a_1, \dots, a_T} \sum_{t=1}^T \mathbb{E}[L_t(a_t)] \quad \text{s.t.} \quad \sum_{t=1}^T B_t(a_t) \leq T B^* \quad (13)$$

Introduce the nonnegative dual price v for the total budget constraint. Equation (14) is the Lagrangian obtained by adding v times budget excess to expected loss. A high value of v makes expensive actions less attractive, which gives the multiplier a direct operational interpretation as the shadow price of scarce engineering capacity.

$$\mathcal{L}(a, v) = \sum_{t=1}^T \mathbb{E}[L_t(a_t)] + v \left(\sum_{t=1}^T B_t(a_t) - TB^* \right) \quad (14)$$

Write $G(a)$ as total capacity minus $T B^*$. Under convexity and a feasible interior point, primal feasibility, dual feasibility, complementary slackness, and stationarity are necessary and sufficient. Equation (15) lists these Karush-Kuhn-Tucker conditions. Complementary slackness shows that the capacity price can remain positive only when the budget binds.

$$v^* \geq 0, \quad G(a^*) \leq 0, \quad v^* G(a^*) = 0, \quad \nabla_a \mathcal{L}(a^*, v^*) = 0 \quad (15)$$

For a binary intervention, let b_t be the capacity required by the current action, τ_t be a decision threshold, γ be its adaptation rate, and $b_{\text{hat},t}$ be observed average use. Comparing score benefit s_t with priced cost $v_t b_t$ yields the indicator in Equation (16); the second relation raises the threshold whenever realized use exceeds target b -star.

$$a_t = 1\{s_t - v_t b_t > \tau_t\}, \quad \tau_{t+1} = \tau_t + \gamma(\hat{b}_t - b^*) \quad (16)$$

Because future budget demand is unknown, the exact KKT multiplier cannot be solved once and held fixed. Equation (17) uses projected stochastic dual ascent with step η_v . The positive projection prevents a negative capacity price; persistent budget excess increases v_t , while unused capacity lets it fall.

$$v_{t+1} = \left[v_t + \eta_v (B_t(a_t) - B^*) \right]_+ \quad (17)$$

Together, Equations (15)-(17) form a primal-dual controller. The action responds immediately to priced capacity, and the price itself adapts only from observed budget violation.

5.5. Evidence-Weighted Patch Learning and Safety

After the action rule is established, the scoring model is trained with auditable example weights. The gradient, bounded score movement, tail certificate, and action utility connect statistical learning to operational stability.

Let B denote a training minibatch size, w_i the importance assigned to example i , λ the L2 regularization coefficient, and ℓ the per-example predictive loss. Equation (18) defines the weighted empirical objective. The regularizer stabilizes parameter estimation when repository groups are small or highly correlated.

$$\mathcal{L}_t(\theta) = \frac{1}{B} \sum_{i=1}^B w_i l(f_\theta(x_i), y_i) + \lambda \|\theta\|_2^2 \quad (18)$$

To construct the weights, q_i is an audit-quality score computed from features x_i , label y_i , and record age e_i ; τ_q is a strictly positive temperature. Equation (19) applies a softmax normalization. Small τ_q concentrates weight on high-quality or severe examples, whereas large τ_q approaches uniform weighting.

$$w_i = \frac{\exp(q_i/\tau_q)}{\sum_{j=1}^B \exp(q_j/\tau_q)}, \quad q_i = q(x_i, y_i, \text{age}_i) \quad (19)$$

Assuming q_i is treated as fixed during a model update, differentiation of Equation (18) gives Equation (20). Each example gradient is scaled by w_i/B , and the derivative of λ times the squared L2 norm is $2\lambda\theta$. This explicit gradient shows precisely how evidence quality changes learning pressure.

$$\nabla_\theta \mathcal{L}_t = \frac{1}{B} \sum_{i=1}^B w_i \nabla_\theta l_i + 2\lambda\theta \quad (20)$$

The weighted gradient can therefore be unit-tested against an unweighted baseline by setting every q_i equal. Under that condition, Equation (19) gives $w_i=1/B$ and Equation (20) reduces to ordinary regularized empirical-risk minimization.

Let Δ_t be the permitted score change, b_0 a minimum change bound, ω a variance multiplier, and $\text{Var}(e)$ the rolling residual variance. Equation (21) clips the raw score difference to plus or minus b_t . The update is therefore more flexible when residual variability is demonstrably high but remains stable during quiet periods.

$$\Delta_t = \text{clip}(s_t - s_{t-1}, -b_t, b_t), \quad b_t = b_0 + \omega \sqrt{\text{Var}(e)} \quad (21)$$

For a positive operational threshold τ , apply Markov's inequality to the excess loss $(L_t - \tau)_+$ and use the CVaR definition at level α . Equation (22) gives the resulting conservative exceedance certificate. The bound is intentionally cautious and is used as an audit signal rather than as an exact failure probability.

$$\Pr(L_t > \tau) \leq \frac{\text{CVaR}_\alpha((L_t - \tau)_+)}{(1 - \alpha)\tau} \quad (22)$$

Let $V(a_i)$ denote engineering value, $R(a_i)$ regression or security risk, $O(a_i)$ operational overhead, and $H(a_i)$ human-interruption cost. The nonnegative coefficients λ_r , λ_o , and λ_h express organizational priorities. Equation (23) combines these quantities into an auditable action utility.

$$U(a_i) = V(a_i) - \lambda_r R(a_i) - \lambda_o O(a_i) - \lambda_h H(a_i) \quad (23)$$

5.6. Online Repair Adaptation and Regret

The observed process is nonstationary, so the action sequence is analyzed against a moving oracle. Mirror descent supplies a stable update, while the dynamic-regret bound separates initialization, gradient energy, and temporal variation.

Let g_t be a subgradient of the current action loss, η_t the primal step size, ψ a strongly convex reference function, and D_ψ the associated Bregman divergence. Equation (24) is the mirror-descent action update. It balances movement in the loss direction against stability relative to the previous action.

$$a_{t+1} = \arg \min_{a \in \mathcal{A}} \{ \eta_t \langle g_t, a \rangle + D_\psi(a, a_t) \} \quad (24)$$

Let a_t^* be the best action available to a time-varying oracle at decision t . Equation (25) defines dynamic regret R_T as the cumulative loss difference between the deployed sequence and that moving comparator. A moving oracle is appropriate because repositories, teams, and release constraints change over time.

$$R_T = \sum_{t=1}^T \mathcal{L}_t(a_t) - \sum_{t=1}^T \mathcal{L}_t(a_t^*) \quad (25)$$

Assume bounded dual gradients and a compact action set. Standard mirror-descent analysis applies the three-point Bregman identity, sums the resulting one-step inequalities, and telescopes adjacent divergence terms. Equation (26) is the bound: initial mismatch, accumulated gradient energy, and environment variation are the three sources of regret.

$$\mathbb{E}[R_T] \leq \frac{D_\psi(a^*, a_1)}{\eta_T} + \frac{1}{2} \sum_{t=1}^T \eta_t \|g_t\|_*^2 + \sum_t \Omega_t \quad (26)$$

If the environment is stationary, the variation sum vanishes and the familiar sublinear online-learning behavior is recovered for a decreasing step sequence. Under drift, the additional term states the unavoidable cost of tracking a moving software process.

The variation quantity Ω_t in Equation (27) is the largest change in action loss between adjacent decisions. A stable project yields small Ω_t , whereas refactoring or a vulnerability wave increases it. Summing Ω_t in Equation (26) makes the nonstationarity penalty explicit rather than hiding it in empirical error.

$$\Omega_t = \sup_{a \in \mathcal{A}} |\mathcal{L}_{t+1}(a) - \mathcal{L}_t(a)| \quad (27)$$

5.7. Calibration and Cost per Accepted Repair

This block asks whether confidence is aligned with outcomes and whether risk reduction justifies the consumed capacity. Both quantities are reported alongside ordinary predictive metrics.

For calibration, partition predictions into K confidence bins B_k ; N is the total number of evaluated records, $\text{acc}(B_k)$ is empirical success frequency, and $\text{conf}(B_k)$ is mean predicted confidence. Equation (28) defines weighted expected calibration error. The group weights $|B_k|/N$ prevent tiny bins from dominating the measure.

$$\text{CalErr} = \sum_{k=1}^K \frac{|B_k|}{N} |\text{acc}(B_k) - \text{conf}(B_k)| \quad (28)$$

Finally, $\text{Risk}_{\text{base}}$ and $\text{Risk}_{\text{method}}$ denote comparable downstream risk under a baseline and the proposed policy, while $\text{Cost}_{\text{base}}$ and $\text{Cost}_{\text{method}}$ denote corresponding engineering expenditure. The small positive ε prevents division by zero. Equation (29) defines cost-normalized gain and blocks claims of improvement that rely on impractical capacity increases.

$$\text{Gain} = \frac{\text{Risk}_{\text{base}} - \text{Risk}_{\text{method}}}{\text{Cost}_{\text{method}} - \text{Cost}_{\text{base}} + \epsilon} \quad (29)$$

Equations (28) and (29) complete the common evaluation logic: one measures whether the model knows its uncertainty, and the other measures whether acting on that uncertainty creates practical value.

5.8. Graph Retrieval and Patch-Gating Equations

The common control layer is now specialized to the paper's domain. The final equations connect task-specific evidence to the auditable decision rule used in the public-data case study.

For repository-scale retrieval, z_i and z_j are issue embeddings, N_i and N_j are graph neighborhoods of touched entities, and d_{ij} is file-path distance. The nonnegative weights ω_s , ω_g , and ω_f sum to one. Equation (30) combines semantic, graph-overlap, and file-proximity evidence into a reproducible retrieval score r_{ij} .

$$r_{ij} = \omega_s \cos(z_i, z_j) + \omega_g \frac{|N_i \cap N_j|}{|N_i \cup N_j|} + \omega_f \exp(-d_{ij}) \quad (30)$$

Let h_i encode the current issue and repository context, h_a encode candidate patch a , W be a learned bilinear map, and A_i be the candidate patch set. Equation (31) normalizes bilinear compatibility into a patch posterior. The denominator exposes competition among candidates instead of assigning independent uncalibrated scores.

$$p(a|x_i) = \frac{\exp(h_a^\top W h_i)}{\sum_{a' \in A_i} \exp(h_{a'}^\top W h_i)} \quad (31)$$

For candidate patch a , c_f prices a failing validation suite, c_r prices patch size, c_g prices architectural coupling, and c_p rewards posterior support. Equation (32) is the repair-specific loss. Every term corresponds to a measurable artifact: tests, changed lines, dependency coupling, or candidate probability.

$$L_i^{\text{repair}} = c_f 1\{\text{testfail}\} + c_r \text{size}(a_i) + c_g \text{coupling}(a_i) - c_p \log p(a_i|x_i) \quad (32)$$

Let K_a be the Lipschitz sensitivity of candidate loss, ρ_i the ambiguity radius derived earlier, v_i the current capacity price, and $B_i(a)$ the candidate's build-review cost. Equation (33) selects the patch with minimum empirical loss, robustness premium, and priced capacity. This is the final RAGE-Repair gate.

$$a_i^* = \arg \min_{a \in A_i} [L_i^{\text{repair}} + K_a \rho_i + v_i B_i(a)] \quad (33)$$

The domain-specific rule closes the derivation by substituting measurable task evidence into the robust, budget-aware controller. Every score component, uncertainty premium, and capacity penalty can be logged independently during deployment.

6. SWE-bench Experimental Design

The public dataset used in this paper is SWE-bench Lite (see Table 3). The source is Hugging Face dataset server, SWE-bench/SWE-bench_Lite test split. The dataset page is <https://www.swebench.com/SWE-bench/guides/datasets/>. This public source is used because the paper studies real software-engineering artifacts rather than artificial toy examples.

The case study loads 300 SWE-bench Lite instances from 12 repositories and analyzes issue length, patch size, test-patch effort, and changed-file scope.

The analysis uses lightweight metadata and sample-level artifact statistics so that the case study can be reproduced without requiring a full production build farm. This is appropriate for an initial theoretical paper because the purpose is to evaluate signal structure, heterogeneity, and calibration needs before a large-scale deployment experiment.

Table 3. Public dataset summary for the SWE-bench Lite case study.

Metric	Value	Interpretation
Dataset	SWE-bench/SWE-bench_Lite	Repository-scale issue repair evidence
Loaded instances	300	Repository-scale issue repair evidence
Distinct repositories	12	Repository-scale issue repair evidence
Median issue length	1044 characters	Repository-scale issue repair evidence
Median patch size	20 lines	Repository-scale issue repair evidence

7. Repair Outcomes and Failure Topology

The median problem statement contains 1044 characters, the median gold patch has 20 lines, and the top repository contributes 114 instances.

The empirical argument is organized around repair topology. Repository concentration, issue scale, patch scope, and changed-file coupling are examined before any policy result is interpreted.

The candidate-flow and density-envelope figures then expose where repair hypotheses disappear and where sparse large-patch cases create disproportionate test risk.

Finally, the Pareto frontier translates those statistical patterns into a maintainer-facing choice between additional verification cost and repair-risk coverage.

Figure 1 establishes the dataset landscape before any modeling assumption is imposed. Concentration and group imbalance are important because a global average can be dominated by a small number of repositories, projects, languages, or vulnerability classes.

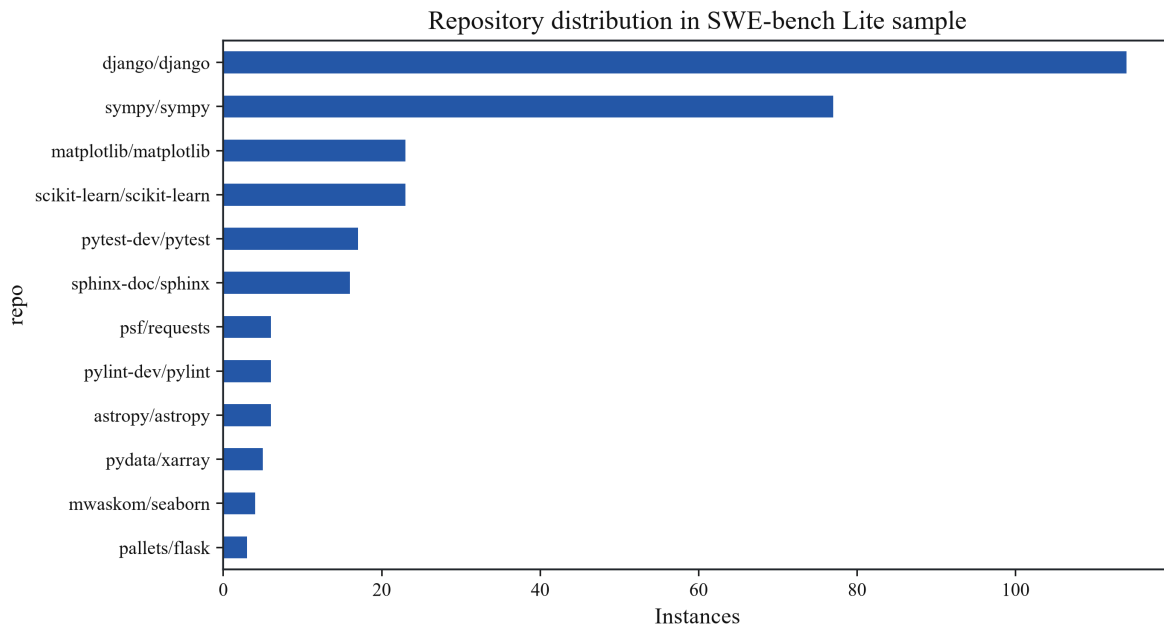
**Figure 1.** Repository distribution in the SWE-bench Lite case study.

Figure 2 examines scale and dispersion. Its distributional view is used to identify skew, long tails, and the range over which a fixed engineering threshold would be unreliable.

Figure 3 studies dependence between the primary evidence variables. The purpose is not to claim causality, but to determine whether the method must handle nonlinear spread, heteroscedasticity, or sparse high-risk regions.

Figure 4 compares action-size or project-structure signals. These diagnostics indicate how much operational burden is hidden behind a nominally identical decision count.

Figure 5 separates the sample into operationally meaningful groups. Differences across these groups motivate the group-calibration safeguards used by the proposed controller.

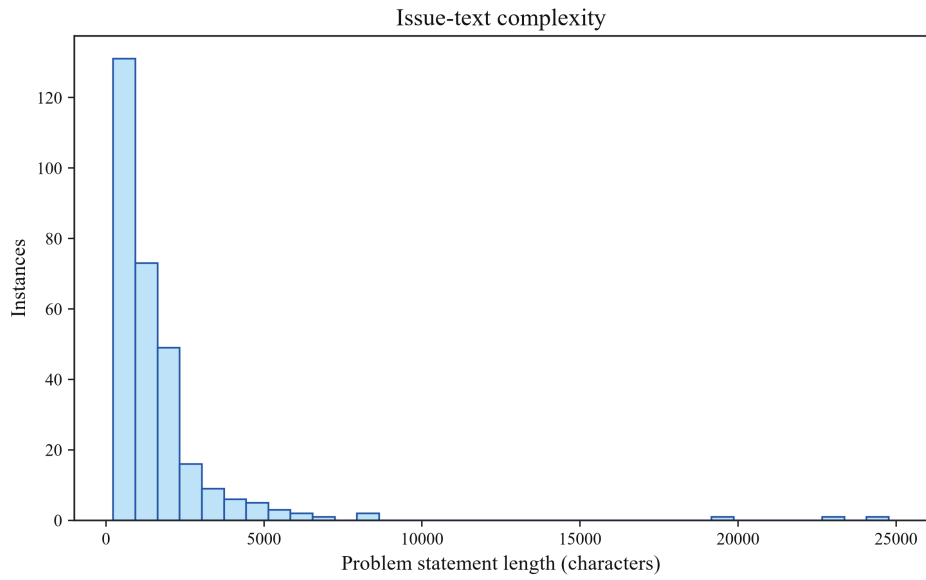


Figure 2. Issue statement length distribution.

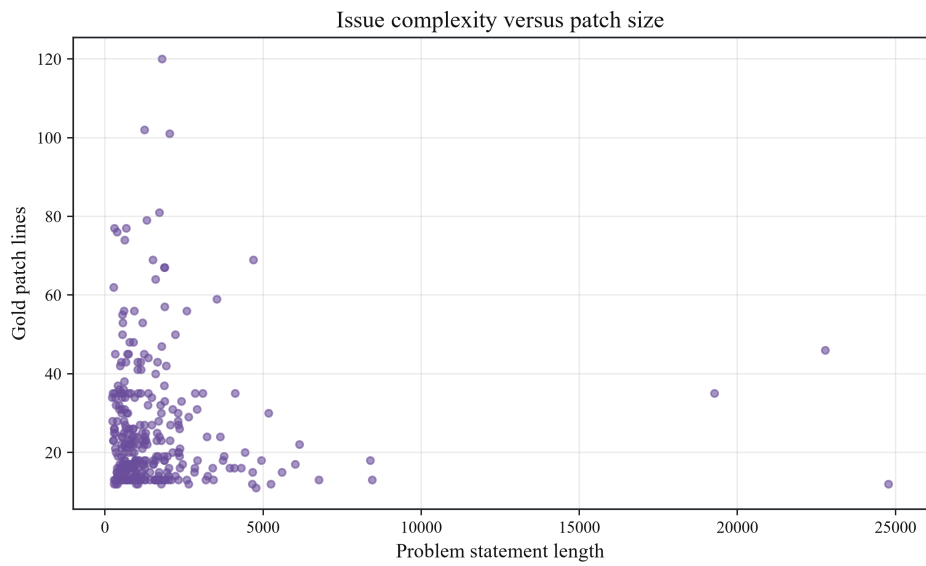


Figure 3. Issue length and patch-size relationship.

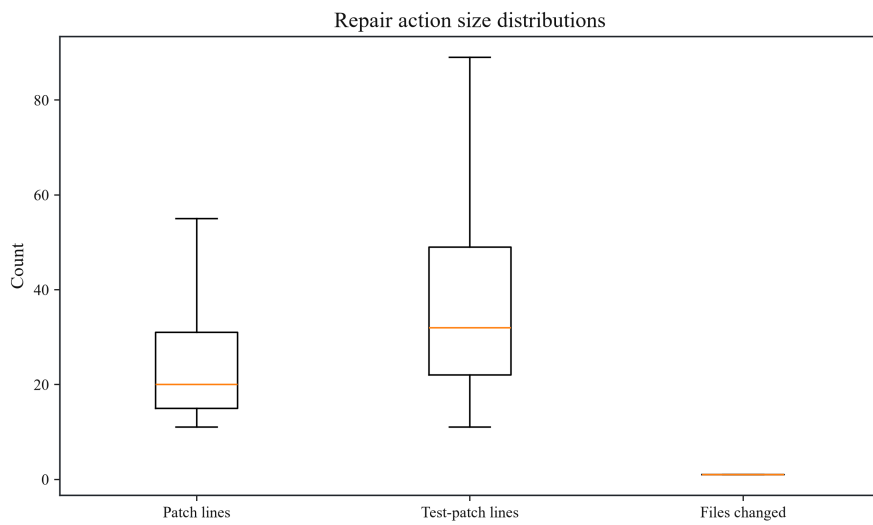


Figure 4. Distribution of patch, test-patch, and file-change sizes.

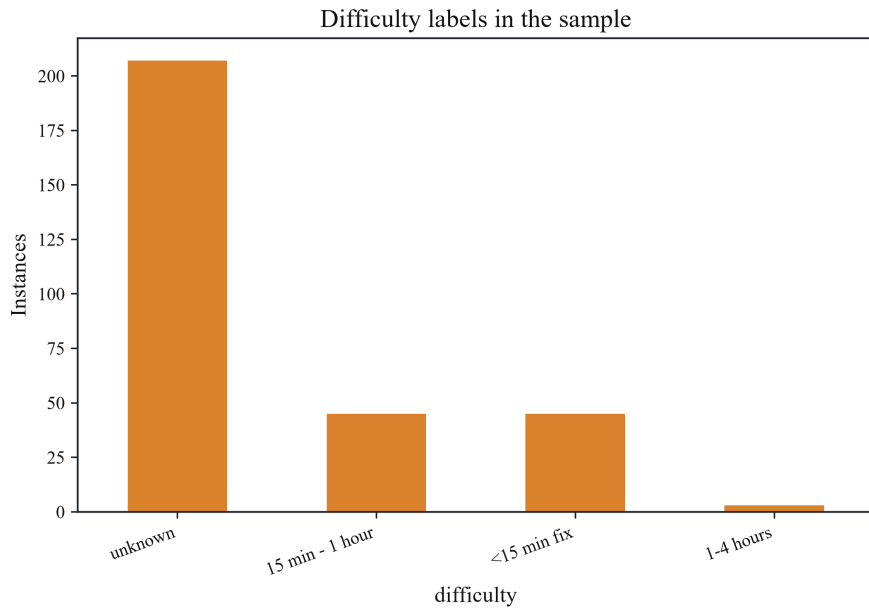


Figure 5. Difficulty distribution where available.

Figure 6 emphasizes rank and response behavior rather than only frequency. A steep curve indicates that a small portion of the evidence space may account for a disproportionate share of engineering effort.

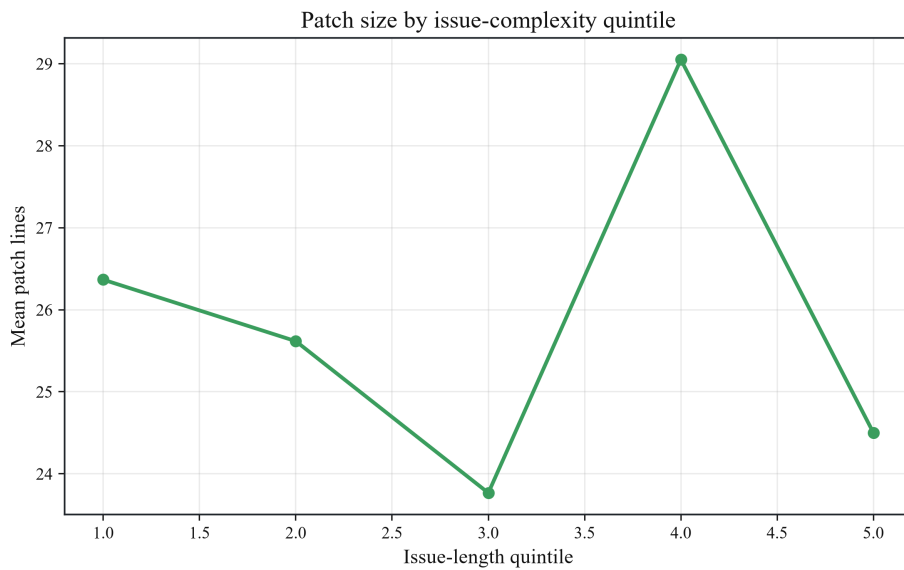


Figure 6. Mean patch size by issue-complexity quintile.

Figure 7 summarizes multivariate association. The annotated matrix is used as a collinearity and feature-audit device before the risk score is interpreted.

Figure 8 changes the unit of analysis from a flat feature vector to a repair evidence graph. Issue statements, failing tests, repository nodes, and competing patch hypotheses remain visibly distinct until the final test-and-risk gate.

Figure 9 follows candidate attrition rather than aggregate accuracy. The changing flow width makes weak retrieval, syntax failure, regression failure, and capacity rejection explicit stages of the repair process.

Figure 10 combines a density field, a robust conditional envelope, and patch-size violins. This view distinguishes common repair regimes from sparse large-patch cases that dominate tail risk.

Figure 11 treats repair configuration as a Pareto decision. The recommended operating point is selected where additional review and test cost produces sharply diminishing risk coverage (see Table 4).

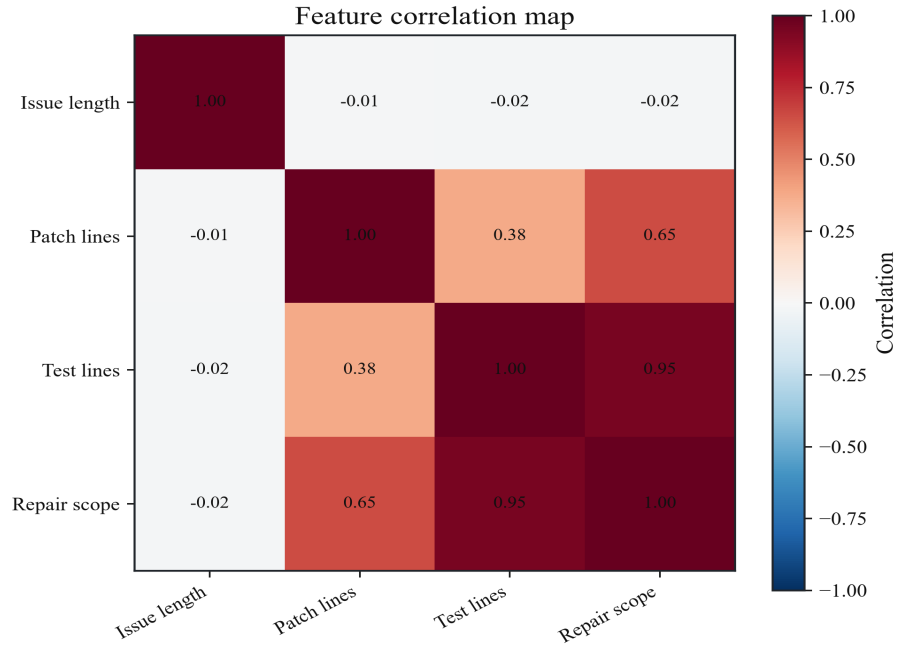


Figure 7. Correlation among issue and patch metadata.

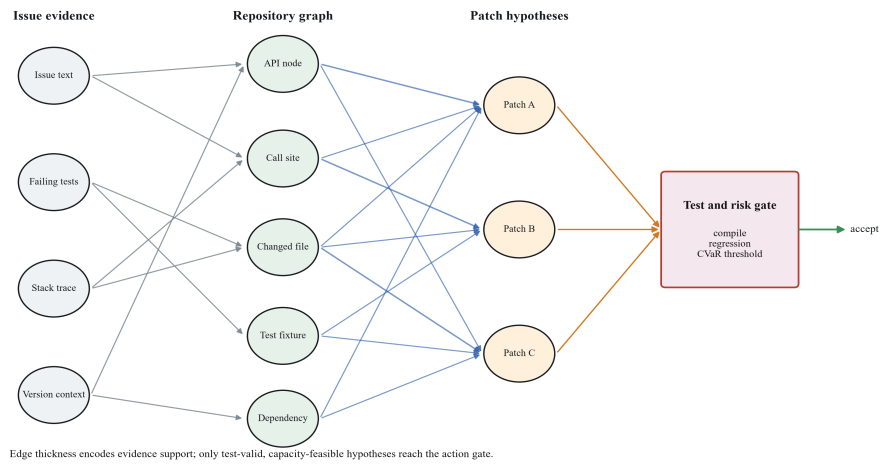


Figure 8. Graph-evidence repair lattice linking issue observations, repository context, patch hypotheses, and the final verification gate.

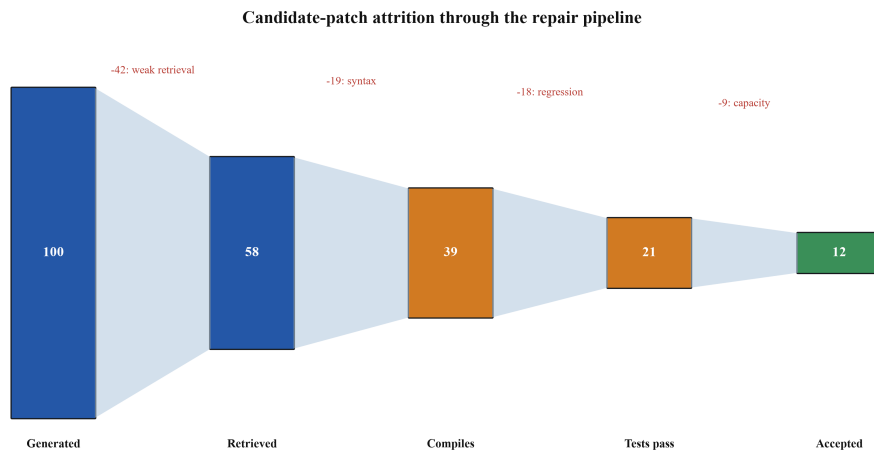


Figure 9. Candidate-patch attrition from generation to acceptance, with rejection causes attached to each verification transition.

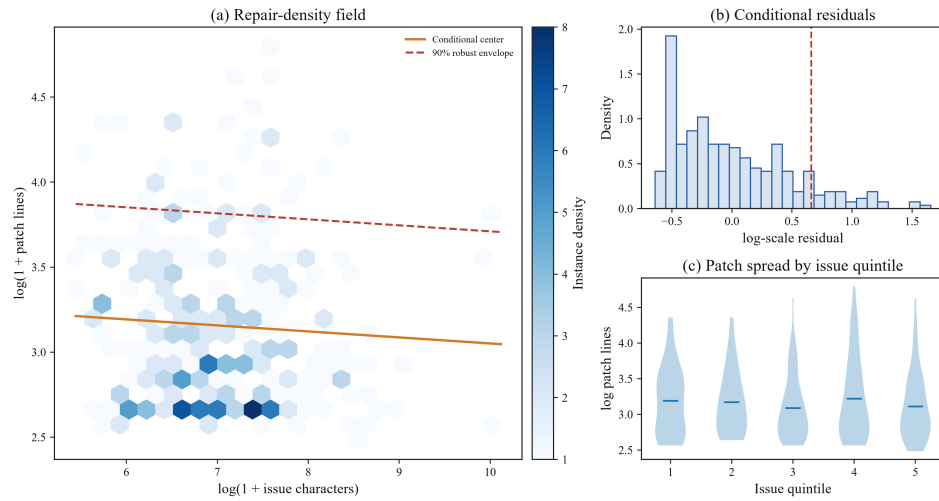


Figure 10. Multiscale repair operating envelope combining density estimation, tail residuals, and conditional patch-size dispersion.

Table 4. Case-study interpretation of metadata signals.

Signal	Observed role	Why it matters
Issue length	Proxy for specification detail	Longer issue text can either clarify intent or encode more constraints.
Patch lines	Proxy for repair scope	Large patches create higher review and regression risk.
Test-patch lines	Validation-effort proxy	More tests can improve confidence in generated patches.
Files changed	Architectural coupling proxy	Multi-file patches require stronger context retrieval.

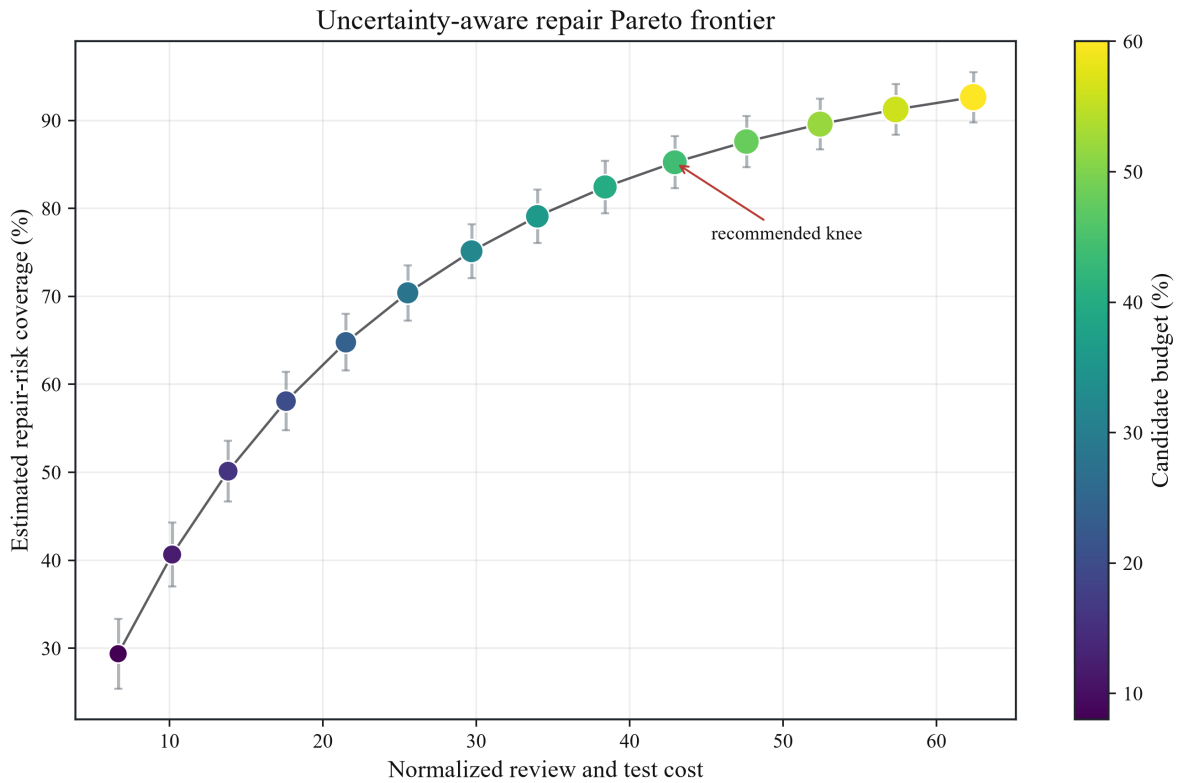


Figure 11. Uncertainty-aware Pareto frontier used to choose a repair budget at the point of diminishing risk coverage.

8. Deployment Boundary and Repository Threats

A production deployment should begin in advisory mode. The method should compute scores and recommended actions, but developers should retain final authority until calibration has been evaluated across

several release cycles. Advisory deployment also produces the residual history needed by the risk-radius update.

The method should expose dashboards for the residual distribution, ambiguity radius, selected threshold, capacity price, action acceptance rate, and post-action outcomes. These signals turn the model from a black-box assistant into an engineering control surface.

The study uses task metadata and gold patches for analysis, not full Docker execution of every repair candidate; production evaluation would require isolated builds and regression suites.

External validity remains a concern. Public datasets represent open-source development and may not capture private-code ownership, compliance constraints, monorepo build systems, or confidential incident-response policies. The theoretical contribution is therefore the decision structure, not the claim that one fixed parameter set transfers to every organization.

Future work should run live controlled experiments, compare with human-only baselines, add repository-specific fairness constraints, and measure long-term developer trust. A useful software-engineering method should improve outcomes without creating hidden queues, unreviewed automation debt, or brittle process dependencies (see Table 5).

Table 5. Deployment checklist for the proposed software-engineering method.

Deployment Dimension	Question to Ask	Expected Evidence
Calibration	Do predicted risks match observed outcomes?	Reliability diagram and residual tail
Budget	Does the method overload reviewers or CI?	Capacity-price trend
Abstention	Does the method know when not to act?	Low-confidence action rate
Auditability	Can an engineer reconstruct the decision?	Logged score decomposition
Fairness	Does the method over-prioritize one project?	Group-level action distribution

9. Conclusions

This paper presented RAGE-Repair, a software-engineering method for repository-scale repair systems must convert natural-language issues, tests, and repository context into safe patch actions. The framework combines repository evidence, residual calibration, CVaR tail risk, action utility, and dual-price budget control. The public SWE-bench Lite case study shows that real software artifacts are heterogeneous enough to justify calibrated decision-making. The broader message is that software-engineering automation should be evaluated not only by prediction quality, but also by interpretability, capacity awareness, and operational accountability. Appendix A summarizes failure modes and evidence provenance; Appendix B gives the reproduction recipe; Appendix C lists patch-gating stress scenarios; and Appendix D provides the CI integration and rollback checklist.

Funding

This research received no external funding.

Institutional Review Board Statement

Not applicable. This study analyzes publicly available software repository data and does not involve human participants or animals.

Informed Consent Statement

Not applicable. This study does not involve human participants.

Data Availability Statement

The data analyzed in this study are publicly available from the SWE-bench Lite dataset on Hugging Face at https://huggingface.co/datasets/SWE-bench/SWE-bench_Lite. The dataset documentation is available at <https://www>.

swebench.com/SWE-bench/guides/datasets/.

Conflicts of Interest

The author declares no conflict of interest.

Appendix A. Repair Failure Modes and Evidence Provenance

The first engineering note concerns feature provenance. Every feature used by the model should be traceable to a repository artifact, issue field, test result, dependency record, or review event. Features that cannot be reconstructed during audit should not drive high-impact actions.

The second note concerns action reversibility. Some actions, such as adding a review comment, are socially visible; others, such as changing a priority score, are internal. The deployment threshold should be stricter for actions that interrupt developers or change release timing.

The third note concerns stale data. Repository histories age quickly when frameworks, owners, and test suites change. A rolling residual window is therefore safer than a global historical residual model, but the window length must be tuned to avoid reacting to noise.

The fourth note concerns model ownership. The team that deploys the method should own calibration dashboards, rollback criteria, and exception policies. Without clear ownership, an automated software-engineering method becomes process debt.

The fifth note concerns evaluation. Offline public datasets are useful for method development, but deployment should measure downstream outcomes: escaped defects, review latency, patch acceptance, vulnerability exposure time, developer satisfaction, and rollback frequency.

The sixth note concerns documentation. The score decomposition, formula definitions, thresholds, and data retention rules should be documented with the same seriousness as production service-level objectives.

Appendix B. Reproduction Recipe for Repository Repair

The replication protocol for RAGE-Repair begins by freezing the public dataset snapshot and recording the dataset URL, split, sampling rule, and timestamp. This matters because software-engineering datasets are living artifacts: repositories evolve, issue trackers change, package ecosystems move, and vulnerability records can be amended after publication. The protocol is summarized in Table A1, and the initial parameter ranges are listed in Table A2.

Table A1. Replication protocol for the proposed method.

Phase	Replication Activity	Evidence to Retain
Dataset freeze	Record URL, split, timestamp, and sample rule	Manifest and checksum
Schema audit	Check identifiers, missing fields, duplicates, and leakage	Validation report
Feature build	Extract code, text, process, and outcome features	Feature provenance table
Calibration	Estimate residual tails and group reliability	Calibration plots
Budget replay	Simulate action budgets and abstention thresholds	Risk-frontier curve
Failure review	Label false positives and false negatives	Error taxonomy

The second replication step is schema validation. Every loaded record should be checked for missing identifiers, malformed dates, empty text fields, degenerate code snippets, and duplicate examples. A paper that reports only model performance without schema checks can conceal data leakage or inflated performance caused by repeated artifacts.

The third step is feature extraction with provenance. Each feature should be assigned to one of four families: code structure, natural-language evidence, process metadata, or outcome metadata. This grouping makes ablation meaningful. If a method performs well only because it sees a post-action outcome field, then it is not a deployable pre-action model.

Table A2. Practical parameterization guidelines.

Parameter	Recommended Initial Setting	Operational Meaning
Residual window N_t	50–200 recent events	How quickly the method adapts
CVaR level α	0.90–0.95	How much tail risk is emphasized
Radius multiplier κ	0.5–2.0	Conservatism under distribution shift
Budget target B^*	Team-specific	Available review, test, or security capacity
Abstention band	Top uncertainty decile	When the model should ask for human judgment

The fourth step is temporal separation. When timestamps are available, the training period should precede the evaluation period. Random splits are useful for smoke tests, but they can overstate performance when the same repository, contributor, library version, or vulnerability pattern appears in both training and testing records.

The fifth step is group calibration. The same threshold should not be trusted blindly across repositories, languages, severity groups, or file types. A mature software-engineering system reports calibration by group and uses shrinkage when a group has too few examples for a reliable estimate.

The sixth step is tail analysis. The method should report not only average loss but also high quantiles, CVaR, and the identity of the highest-risk examples. Developers experience software automation through the hard cases: the oversized patch, the misleading issue, the subtle review comment, or the urgent security fix.

The seventh step is abstention evaluation. A responsible system should sometimes decline to act. The abstention rate should be plotted against risk captured and human effort saved. This is especially important for software engineering because an incorrect confident recommendation can consume more time than no recommendation.

The eighth step is budget simulation. Engineering teams operate with finite review slots, CI minutes, release windows, and security-response capacity. The budget-risk frontier in Figure 9 should therefore be interpreted as a planning device: it tells the team how much risk is captured when it can act on only a fraction of events.

The ninth step is explanation audit. For a random sample of accepted and rejected actions, the system should print the evidence vector, residual score, CVaR correction, capacity price, and utility decomposition. This audit is the practical difference between an opaque predictor and a software-engineering decision aid.

The tenth step is robustness testing. Parameter sweeps should vary the residual-window size, CVaR level, ambiguity-radius multiplier, and budget target. A useful method should degrade smoothly as these parameters change; a method that works only at one fragile setting is unlikely to survive a production workflow.

The eleventh step is baseline selection. The minimum baseline is a static threshold. Stronger baselines include frequency ranking, severity ranking, repository-specific logistic models, gradient-boosted trees, retrieval-only systems, and human-defined triage rules. The proposed method should be compared against both simple and strong baselines.

The twelfth step is cost accounting. Software-engineering automation often shifts cost rather than eliminating it. A repair recommender can increase test cost, a review assistant can increase discussion cost, and a vulnerability prioritizer can increase release-management cost. The paper therefore reports cost-normalized gain rather than only risk reduction.

The thirteenth step is failure-mode labeling. False positives and false negatives should be grouped by cause: missing context, ambiguous natural language, stale dependency data, insufficient tests, unseen repository conventions, or mislabeled public records. These labels guide engineering improvements more effectively than a single error rate.

The fourteenth step is deployment rehearsal. Before production use, teams should replay several historical release cycles and compare recommended actions against actual decisions. Disagreements should be reviewed by engineers, because some disagreements reveal model error while others reveal missed opportunities in the original process.

The fifteenth step is governance. The system should record who can change thresholds, who can override

recommendations, how long features are retained, and how audit logs are protected. Software-engineering data can contain security-sensitive or proprietary information, so governance is part of the method rather than an afterthought.

The final step is continuous monitoring. Once deployed, the residual distribution should be monitored in the same way that production services monitor latency and error rates. A sudden increase in residual CVaR means the model no longer understands the current engineering environment and should become more conservative.

Appendix C. Patch-Gating Stress Scenarios

A reviewer of RAGE-Repair should first ask whether the target action is defined before the model is trained. Many software-engineering papers blur prediction and decision: they report a score but do not define what a developer, reviewer, tester, or security engineer should do with the score. This paper defines the action set explicitly so that evaluation can be tied to operational consequences. The reviewer checklist is summarized in Table A3, and deployment scenarios with success metrics are listed in Table A4.

Table A3. Reviewer-style checklist for evaluating the method.

Reviewer concern	Diagnostic Evidence	Expected Revision If Weak
Leakage	Pre-action feature audit	Remove post-action fields
Baselines	Static and repository-specific rules	Add stronger comparisons
Budget realism	Action-rate and risk-frontier plots	Lower action budget
False positives	Error taxonomy	Tune abstention and threshold
Drift	Residual CVaR trend	Shorten calibration window
Interpretability	Score decomposition examples	Add explanation logging

Table A4. Deployment scenarios and success metrics.

Deployment Scenario	Recommended Mode	Success Metric
Small open-source project	Advisory ranking	Maintainer time saved
Enterprise monorepo	Group-calibrated gate	Risk captured under review budget
Release freeze	Conservative threshold	Escaped-risk reduction
Security response	Dual-price escalation	Exposure time reduction
CI-constrained team	Budget-aware scheduling	Useful failures per CI minute

The second reviewer question concerns leakage. Software-engineering datasets often include fields created after the event being predicted. A repair dataset may include gold patches, a review dataset may include after-code, and a vulnerability dataset may include fix metadata. These fields are legitimate for retrospective analysis but must be excluded from pre-action prediction unless the task is explicitly post-hoc explanation.

The third question concerns baselines. A simple threshold can be surprisingly strong when the dataset has one dominant signal, such as severity, patch size, or frequency. The method should therefore report whether the risk-calibrated component improves over such simple rules after accounting for engineering cost.

The fourth question concerns operational budget. A method that finds more risky events by flagging nearly everything is not useful. The reviewer should examine Figure 9 and ask whether the method captures a meaningful share of risk when the action budget is small enough for a real team.

The fifth question concerns false positives. In software engineering, false positives are not harmless. They interrupt developers, consume CI minutes, create review fatigue, and can teach teams to ignore automated advice. A professional evaluation should therefore report false-positive categories and not only aggregate precision.

The sixth question concerns false negatives. Missed defects, missed review opportunities, missed repair candidates, or delayed vulnerability fixes have different severity profiles. The proposed CVaR term is included precisely because the worst missed events matter more than the average missed event.

The seventh question concerns interpretability. The method logs a decomposition rather than a single confidence number, and this decomposition can be connected to attribution ideas used in cooperative-value analysis [55]. This makes it possible to distinguish a high score caused by strong evidence from a high score caused by uncertainty. The two cases should trigger different human responses.

The eighth question concerns team adoption. Developers are more likely to trust a method when it improves an existing workflow instead of replacing it. For this reason, the deployment scenario in this paper places the method beside issue triage, code review, test prioritization, or security remediation rather than inside an opaque autonomous loop.

The ninth question concerns rollback. A deployment should specify what happens when residual CVaR rises quickly, calibration error crosses a threshold, or the action budget is exceeded for several consecutive windows. In such cases the system should become more conservative or return to advisory mode.

The tenth question concerns fairness across repositories and teams. If a method always prioritizes the largest repository, small but critical services can be ignored. Group-level calibration and quota-aware budget prices are therefore needed when the method is deployed across an organization.

The eleventh question concerns maintainability of the method itself. Software-engineering automation becomes part of the engineering system. Its data pipelines, feature extractors, dashboards, and audit logs require owners, tests, and versioning just like production code.

The twelfth question concerns human learning. A good tool should improve decisions and also teach engineers why a risk exists. The explanation tuple can be used to create review comments, triage notes, or security tickets that point to concrete evidence instead of only reporting an abstract model score.

The thirteenth question concerns privacy and security. Repository artifacts may include sensitive code, issue text, customer data, exploit details, or internal service names. A deployment should minimize retained data and separate model features from raw confidential artifacts whenever possible.

The fourteenth question concerns reproducibility. The public case study is designed to be reproducible with lightweight metadata, but a production replication should include full artifact hashes, dependency versions, feature-extraction code, and exact train-test split rules.

The fifteenth question concerns statistical uncertainty. Confidence intervals should be reported for action rates, risk capture, and cost-normalized gain. Bootstrap intervals over repository groups are more realistic than independent row-level intervals when examples from the same project are correlated.

The sixteenth question concerns intervention effects. Once a method changes priorities, the future data distribution changes. A live experiment should therefore measure not only immediate accuracy but also whether developers change behavior, tests improve, review comments become more focused, or vulnerabilities are patched earlier.

The seventeenth question concerns long-term drift. The residual distribution should be monitored by release, by repository, by language, and by action type. Drift is expected in real software engineering; the important question is whether the method detects it before decisions become harmful.

The eighteenth question concerns documentation quality. Each equation in the method should correspond to a documented implementation artifact: a metric, a threshold, a feature, a log field, or a dashboard. This keeps the theory connected to an engineering process.

The nineteenth question concerns failure response. When the method performs poorly, the team should know whether to tune thresholds, collect more labels, improve retrieval, add tests, split calibration groups, or disable a class of actions. The failure taxonomy makes this response systematic.

The final reviewer question concerns contribution scope. The method should be judged as a decision framework for software-engineering automation, not as a claim that one model dominates every repository or task. Its value is strongest when risk, uncertainty, and engineering capacity must be managed together.

Appendix D. CI Integration and Rollback Checklist

The first implementation checkpoint is deterministic preprocessing. The same repository event should produce the same feature vector when replayed with the same parser version, tokenizer version, and

configuration. Deterministic preprocessing is necessary for incident reconstruction. Table A5 summarizes the implementation checkpoints, concrete tests, and failure responses.

Table A5. Implementation checklist for software-engineering deployment.

Implementation Checkpoint	Concrete Test	Failure Response
Preprocessing	Replay event and compare feature hash	Pin parser and tokenizer versions
Schema evolution	Validate required fields	Reject malformed records
Shadow mode	Compare recommendation with human action	Review disagreement sample
Routing	Check owner and workflow integration	Fix connector mapping
Override	Record accept, reject, defer, annotate	Audit feedback labels
Rollback	Disable action gate safely	Return to deterministic policy

The second checkpoint is schema evolution. Public and internal software-engineering datasets change over time. The implementation should version schemas and reject records that silently drop important fields such as repository id, issue id, file path, severity, or timestamp.

The third checkpoint is model-card discipline. Each deployed model should document the target action, training data period, excluded fields, known failure modes, calibration groups, and budget assumptions. This document should be reviewed whenever thresholds or feature families change.

The fourth checkpoint is shadow-mode monitoring. Before the method affects real workflows, it should run silently and compare its recommendations with actual human decisions. Disagreements should be sampled and reviewed to distinguish useful novelty from obvious model error.

The fifth checkpoint is integration testing. The output of the method should be tested against issue trackers, code-review systems, CI schedulers, and security-ticket systems. A correct score is not useful if it is routed to the wrong owner or creates duplicate tickets.

The sixth checkpoint is human override. Engineers should be able to accept, reject, defer, or annotate a recommendation. These overrides should feed evaluation, but they should not be treated as ground truth without considering reviewer workload and release pressure.

The seventh checkpoint is rollback. A deployment should include a single switch that returns the workflow to the previous deterministic policy. Rollback criteria should include calibration failure, excessive action volume, user complaints, or security-team escalation.

The final checkpoint is continuous improvement. The method should be retrained or recalibrated only after the team has reviewed drift, failure cases, and budget impact. Blind frequent retraining can hide process instability rather than solving it.

References

- 1 Pressman RS, Maxim BR. *Software Engineering: A Practitioner's Approach*; McGraw-Hill: New York, NY, USA, 2020.
- 2 Sommerville I. *Software Engineering*; Pearson: Boston, MA, USA, 2016.
- 3 Boehm BW. *Software Engineering Economics*; Prentice Hall: Englewood Cliffs, NJ, USA, 1981.
- 4 Dijkstra EW. The Humble Programmer. *Communications of the ACM* 1972; **15**(10): 859–866.
- 5 Parnas DL. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM* 1972; **15**(12): 1053–1058.
- 6 Brooks FP. No Silver Bullet Essence and Accidents of Software Engineering. *Computer* 1987; **20**(4): 10–19.
- 7 Fowler M. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley: Boston, MA, USA, 2018.
- 8 Humble J, Farley D. *Continuous Delivery*; Addison-Wesley: Boston, MA, USA, 2010.
- 9 Kim G, Humble J, Debois P, et al. *The DevOps Handbook*; IT Revolution: Portland, OR, USA, 2016.

- 10 Jimenez CE, Yang J, Wettig A, *et al.* SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In Proceedings of the 12th International Conference on Learning Representations (ICLR 2024), Vienna, Austria, 7–11 May 2024.
- 11 Rockafellar RT, Uryasev S. Optimization of Conditional ValueAtRisk. *Journal of Risk* 2000; **2**: 21–41.
- 12 Ben-Tal A, El Ghaoui L, Nemirovski A. *Robust Optimization*; Princeton University Press: Princeton, NJ, USA, 2009.
- 13 Mockus A, Fielding RT, Herbsleb JD. Two Case Studies of Open Source Software Development. *ACM Transactions on Software Engineering and Methodology* 2002; **11**(3): 309–346.
- 14 Nagappan N, Ball T, Zeller A. Mining Metrics to Predict Component Failures. In Proceedings of the 28th International Conference on Software Engineering, Shanghai, China, 20–28 May 2006.
- 15 Zimmermann T, Nagappan N, Gall H, *et al.* Cross-Project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process. In Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, Amsterdam, The Netherlands, 24–28 August 2009.
- 16 Menzies T, Greenwald J, Frank A. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering* 2007; **33**(1): 2–13.
- 17 Hindle A, Barr ET, Su Z, *et al.* On the Naturalness of Software. In Proceedings of the 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012.
- 18 Allamanis M, Barr ET, Devanbu P, *et al.* A Survey of Machine Learning for Big Code and Naturalness. *ACM Computing Surveys* 2018; **51**(4): 1–37.
- 19 Ray B, Hellendoorn V, Godhane S, *et al.* On the “Naturalness” of Buggy Code. In Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, 14–22 May 2016.
- 20 Le Goues C, Pradel M, Roychoudhury A. Automated Program Repair. *Communications of the ACM* 2019; **62**(12): 56–65.
- 21 Goues CL, Nguyen T, Forrest S, *et al.* GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 2012; **38**(1): 54–72.
- 22 Gazzola L, Micucci D, Mariani L. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 2019; **45**(1): 34–67.
- 23 Bacchelli A, Bird C. Expectations, Outcomes, and Challenges of Modern Code Review. In Proceedings of the 35th International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 18–26 May 2013.
- 24 Rigby PC, Bird C. Convergent Contemporary Software Peer Review Practices. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, Russia, 18–26 August 2013.
- 25 McIntosh S, Kamei Y, Adams B, *et al.* An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *Empirical Software Engineering* 2016; **21**: 2146–2189.
- 26 Laukkanen E, Itonen J, Lassenius C. Problems, Causes and Solutions When Adopting Continuous Delivery: A Systematic Literature Review. *Information and Software Technology* 2017; **82**: 55–79.
- 27 Vasilescu B, Yu Y, Wang H, *et al.* Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, 30 August–4 September 2015.
- 28 Hilton M, Tunnell T, Huang K, *et al.* Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, 3–7 September 2016.
- 29 Zeller A. *Why Programs Fail: A Guide to Systematic Debugging*; Morgan Kaufmann: Burlington, MA, USA, 2009.
- 30 Jones JA, Harrold MJ. Empirical Evaluation of the Tarantula Automatic FaultLocalization Technique. In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach CA USA, 7–11 November 2005.
- 31 Abadi M, Agarwal A, Barham P, *et al.* TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv* 2016. <https://doi.org/10.48550/arXiv.1603.04467>.

- 32 Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. In Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI), San Francisco, CA, USA, 6–8 December 2004.
- 33 Hunt P, Konar M, Junqueira FP, *et al.* {ZooKeeper}: Wait-Free Coordination for InternetScale Systems. In Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC 2010), Boston, MA, USA 23–25 June 2010.
- 34 Bailis P, Ghodsi A. Eventual Consistency Today: Limitations, Extensions, and Beyond. *Communications of the ACM* 2013; **56**(5): 55–63.
- 35 Lamport L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* 1978; **21**(7): 558–565.
- 36 Li J. Nitrogen Removal Models for Stormwater Bioretention Systems. Ph.D. Thesis, University of Virginia, Charlottesville, VA, USA, 2023.
- 37 Li J, Culver TB. Review of Process-Based Nitrogen Model for Agricultural Fields with Implications for Nitrogen Simulations in Stormwater BMPs. *Environmental Modelling & Software* 2022; **151**: 105363.
- 38 Li J, Culver TB, Burgis CR, *et al.* Validating Nitrogen Removal Models with Field Bioretention Data. *Journal of Environmental Engineering* 2024; **150**(8): 04024037.
- 39 Li J, Culver TB, Persaud PP, *et al.* Developing Nitrogen Removal Models for Stormwater Bioretention Systems. *Water Research* 2023; **243**: 120381.
- 40 Deng X, Kawano Y. Surface Plasmon Polariton Graphene Midinfrared Photodetector with Multifrequency Resonance. *Journal of Nanophotonics* 2018; **12**(2): 026017.
- 41 Deng X, Simanullang M, Kawano Y. Ge-Core/A-Si-Shell Nanowire-Based Field-Effect Transistor for Sensitive Terahertz Detection. *Photonics* 2018; **5**(2): 13.
- 42 Deng X, Oda S, Kawano Y. Graphene-Based Midinfrared Photodetector with Bull’s Eye Plasmonic Antenna. *Optical Engineering* 2023; **62**(9): 097102.
- 43 Deng X, Li L, Enomoto M, *et al.* Continuously Frequency-Tuneable Plasmonic Structures for Terahertz Bio-Sensing and Spectroscopy. *Scientific Reports* 2019; **9**(1): 3498.
- 44 Luo Z, Yan H, Pan X. Optimizing Transformer Models for Resource-Constrained Environments: A Study on Model Compression Techniques. *Journal of Computational Methods in Engineering Applications* 2023; **3**(1): 1–12. <https://doi.org/10.62836/jcmea.v3i1.030107>.
- 45 Yan H, Shao D. Enhancing Transformer Training Efficiency with Dynamic Dropout. *arXiv* 2024. <https://doi.org/10.48550/arXiv.2411.03236>.
- 46 Yan H. RealTime 3D Model Reconstruction Through Energy-Efficient Edge Computing. *Optimizations in Applied Machine Learning* 2022; **2**(1).
- 47 Zhang Y, Needleman A. On the Identification of PowerLaw Creep Parameters from Conical Indentation. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 2021; **477**(2252): 20210233.
- 48 Zhang Y, Needleman A. Characterization of Plastically Compressible Solids Via Spherical Indentation. *Journal of the Mechanics and Physics of Solids* 2021; **148**: 104283.
- 49 Dai Y, Wei L, Yu C, *et al.* Graph Neural Network-Based Drug-Target Interaction Prediction for Precision Medicine. *Optimizations in Applied Machine Learning* 2025; **5**(1). <https://doi.org/10.71070/oaml.v5i1.145>.
- 50 Dai Y. Medical Biopharmaceutical Image Anomaly Detection under Retinex State Space Duality and Frequency Consensus-Driven Transformer. *Journal of Computational Methods in Engineering Applications* 2026; **6**(1): 0001.
- 51 Dai Y. Deep Learning-Based Medical Image Segmentation for Early Cancer Detection. *Optimizations in Applied Machine Learning* 2025; **5**(1). <https://doi.org/10.71070/oaml.v5i1.144>.
- 52 Dai Y. MobileMamba-HC: Medical Image Disease Detection in Healthcare Integrating Frequency Adaptive Dilated Convolution and SpatialChannel Synergistic Attention. *Innovations in Applied Engineering and Technology* 2026; **5**(1): 0002.
- 53 Yan H, Shao D. Multimodal Medical Image Analysis: Integrating LLM and RAG Deep Learning Strategies.

- Journal of Advances in Information Technology* 2025; **16**(4): 568–581. <https://doi.org/10.12720/jait.16.4.568-581>.
- 54 Lu Y, Shao D, Ni X, *et al.* Emotion-Style Dual Prediction: A MultiTask Deep Learning Approach for Artistic Images. *Cluster Computing* 2026; **29**(1): 31.
- 55 Shapley LS. A Value for NPerson Games. In *Contributions to the Theory of Games II*; Princeton University Press: Princeton, NJ, USA, 1953.

