https://doi.org/10.62836/jcmea.v6i1.0004

Global Science Publishing

JCMEA

Journal of Computational Methods in Engineering Applications

https://ojs.sgsci.org/journals/jcmea

Article

Hierarchical Bayesian Defect Localization under Test and Inspection Constraints

Haitao Gong

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

Abstract: Defect localization is difficult to transfer across projects because test coverage, defect history, and issue-report availability induce different ranking uncertainty. This paper introduces TDL-Prioritize, a hierarchical empirical-Bayes localization model studied on Defects4J. The case study downloads active bug metadata from the official Defects4J repository and obtains 854 active bugs across 17 Java projects. Project priors shrink sparse suspiciousness estimates, a CVaR term protects against hard-to-localize tails, and a capacity price determines how many files can be inspected. The largest project slice is Closure with 174 active bugs, while external issue-report coverage is 97.9% across the loaded metadata. Posterior ridgelines, credible intervals, and rank-migration diagnostics are used instead of a single aggregate score. The derivation links spectrum evidence, posterior variance, robust risk, and the final ranking gate in a form that can be audited by testing teams.

Keywords: software engineering; Defects4J; fault localization; test-driven repair; empirical Bayes

1. Diagnostic Motivation and Study Questions

Fault localization is a ranking problem with pronounced project heterogeneity. The same suspiciousness score can imply very different inspection effort when test coverage, project size, reporting quality, and historical defect density change [1 – 9]. The study therefore begins from hierarchical uncertainty rather than from a universal ranking threshold.

The target problem is fault localization must rank likely repair locations using failing tests, revision evidence, and project-specific historical defect structure. 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 TDL-Prioritize, whose central contribution is a project-calibrated empirical-Bayes and CVaR risk framework for ranking defects before repair actions are attempted. 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 Defects4J dataset [10]. The case study downloads active bug metadata from the official Defects4J repository and obtains 854 active bugs across 17 Java projects. Because the dataset is

Received: 1 October 2026; Accepted: 16 October 2026.

Corresponding: Haitao Gong (gonght82@gmail.com)

public, the study can be audited by other researchers. Because the dataset reflects real software artifacts, it avoids the clean-room assumptions that often make software-engineering methods look stronger than they are.

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 largest project slice is Closure with 174 active bugs, while external issue-report coverage is 97.9% across the loaded metadata. 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 reviews spectrum-based localization, Section 3 introduces the hierarchical project model, Sections 4 and 5 derive posterior ranking under capacity constraints, and Sections 6–8 report the Defects4J cohort, rank diagnostics, and benchmark limitations. Section 9 closes the argument.

2. Spectrum-Based Localization and the Bayesian Gap

Classical software engineering emphasizes modularity, requirements clarity, verification, maintainability, and economic trade-offs [1–10]. These ideas remain central even when the implementation uses machine learning or repository mining. A model that cannot be explained to engineers is difficult to trust during a release freeze or production incident.

Mining software repositories established that commits, issues, tests, code metrics, and social process signals can predict defect and maintenance outcomes [13–19]. However, prediction alone is not enough. Engineering decisions consume scarce human and compute resources, so a useful method must link predicted risk to action budgets.

Research on automated repair, debugging, fault localization, code review, and continuous integration provides operational context [20–30]. These areas share a common pattern: many candidate actions are available, only a few can be checked carefully, and mistakes are costly when they reach production.

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]. Even when this paper studies a local repository-level action, the decision is made inside a distributed development system of build workers, package registries, review queues, and deployment pipelines.

Independent model validation provides a direct analogy for repository-scale evaluation. Field bioretention studies distinguish process-model development from validation against observed operating data [36,37], while the associated dissertation and process-model review make assumptions, transfer limits, and calibration evidence explicit [38, 39]. The present study follows the same logic by separating score construction from temporal backtesting and post-action residual checks.

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, 41]; multifrequency graphene and nanowire transistor detectors likewise show why heterogeneous response channels must be preserved before inference [42,43]. In this paper, repository, test, process, and uncertainty channels are therefore retained separately until the final action gate.

Resource-aware learning is especially relevant to deployable software automation. Energy-efficient edge reconstruction demonstrates that latency and computation belong inside the method definition [44], dynamic dropout treats training effort as an adaptive quantity [45], and compression research makes explicit the trade-off between representation quality and constrained execution [46]. These precedents motivate the capacity price,

abstention rule, and cost-normalized utility used here.

Finally, recent multimodal and multi-task systems show that evidence fusion should preserve modality-specific uncertainty [47, 48]. Medical anomaly detection, segmentation, and mobile disease detection provide examples of multi-stage or frequency-aware representation pipelines [49 – 51], while graph-based drug-target prediction motivates relational evidence aggregation [52]. Complementary inverse-identification studies in indentation mechanics further emphasize parameter identifiability and uncertainty under indirect observations [53, 54]. These ideas support the paper's graph, hierarchical, interaction, or hazard representation without implying that domain-specific results transfer unchanged.

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. Hierarchical Project Model and Notation

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 ν_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
ν_t	capacity price	scarcity of engineering budget
CVaR_α	tail-risk measure	expected loss beyond a high quantile

4. TDL-Prioritize Posterior Ranking Algorithm

The proposed method, TDL-Prioritize, 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 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 (see Table 2).

Table 2. TDL-Prioritize algorithmic procedure.

Line	Operation	Audit Signal
1	Load failing tests, changed revisions, and project metadata.	Bug context
2	Estimate suspiciousness from test and revision signals.	Fault score
3	Apply project-aware empirical Bayes shrinkage.	Cross-project prior
4	Add CVaR residual correction for hard-to-localize bugs.	Tail risk
5	Rank candidate files and patch templates.	Repair queue
6	Record score decomposition for each recommendation.	Developer explanation

5. Posterior Risk and Capacity 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. Localization Risk and Test-Spectrum State

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 test-driven defect localization 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, ell 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.

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

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.

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

5.2. Empirical Residual Law across Projects

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 $\delta_{e,i}$ 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.

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

The symbol W_1 denotes the order-one Wasserstein distance between probability laws, and ρ_t is a nonnegative ambiguity radius. Equation (4) defines $P_t(\rho_t)$ 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.

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

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.

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

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.

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

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 $K_\theta \rho_t$.

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

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.

$$\rho_t = C_x \sqrt{\frac{\log(2/\delta)}{N_t}} + \kappa \widehat{\mathrm{CVaR}}_\alpha(|e|) \tag{8}$$

5.3. Tail Risk for Hard-to-Localize Defects

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)_+ equals 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.

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

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.

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

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.

$$\nabla_\theta \widehat{\mathrm{CVaR}}_\alpha = \frac{1}{N_t(1-\alpha)} \sum_{i=1}^{N_t} 1\left\{ L_i > \zeta^\star \right\} \nabla_\theta L_i \tag{11}$$

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-β.

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

5.4. Capacity-Constrained File Inspection

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.

$$\min_{a_1,\ldots,a_T} \sum_{t=1}^{T} \mathbb{E}\left[L_t(a_t) \right] \quad \text{s.t.} \sum_{t=1}^{T} B_t(a_t) \le TB^\star \tag{13}$$

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

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

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.

$$\nu^\star \ge 0, \quad G(a^\star) \le 0, \quad \nu^\star G(a^\star) = 0, \quad \nabla_a \mathcal{L}(a^\star, \nu^\star) = 0 \tag{15}$$

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 $b_{hat,t}$ be observed average use. Comparing score benefit s_t with priced cost ν_t b_t yields the indicator in Equation (16); the second relation raises the threshold whenever realized use exceeds target b-star.

$$a_t = 1\{ s_t - \nu_t b_t > \tau_t \}, \quad \tau_{t+1} = \tau_t + \gamma\left(\hat{b}_t - b^\star \right) \tag{16}$$

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_ν. The positive projection prevents a negative capacity price; persistent budget excess increases ν_t, while unused capacity lets it fall.

$$\nu_{t+1} = \left[\nu_t + \eta_\nu \left(B_t(a_t) - B^\star \right) \right]_+ \tag{17}$$

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. Project-Weighted Posterior Learning

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 ell 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.

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

To construct the weights, q_i is an audit-quality score computed from features x_i, label y_i, and record age_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.

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

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.

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

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 Delta_t be the permitted score change, b_0 a minimum change bound, ω a variance multiplier, and Vâr(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.

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

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.

$$\Pr\left(L_t > \tau\right) \le \frac{\text{CVaR}_\alpha\left(\left(L_t - \tau\right)_+\right)}{(1-\alpha)\tau} \tag{22}$$

Let $V(a_t)$ denote engineering value, $R(a_t)$ regression or security risk, $O(a_t)$ operational overhead, and $H(a_t)$ 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.

$$U\left(a_t\right) = V\left(a_t\right) - \lambda_r R\left(a_t\right) - \lambda_o O\left(a_t\right) - \lambda_h H\left(a_t\right) \tag{23}$$

5.6. Dynamic Ranking under Test-Suite Drift

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.

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

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.

$$R_T = \sum_{t=1}^{T}\mathcal{L}_t\left(a_t\right) - \sum_{t=1}^{T}\mathcal{L}_t\left(a_t^\star\right) \tag{25}$$

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.

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

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

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

5.7. Posterior Calibration and Search Efficiency

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, $acc(B_k)$ is empirical success frequency, and $conf(B_k)$ is mean predicted confidence. Equation (28) defines weighted expected calibration error. The group weights $|B_k|/N$ prevent tiny bins from dominating the measure.

$$\mathrm{CalErr} = \sum_{k=1}^{K} \frac{\left| B_k \right|}{N} \left| \mathrm{acc}\left(B_k \right) - \mathrm{conf}\left(B_k \right) \right| \tag{28}$$

Finally, $Risk_{base}$ and $Risk_{method}$ denote comparable downstream risk under a baseline and the proposed policy, while $Cost_{base}$ and $Cost_{method}$ 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.

$$\mathrm{Gain} = \frac{\mathrm{Risk}_{base} - \mathrm{Risk}_{method}}{\mathrm{Cost}_{method} - \mathrm{Cost}_{base} + \epsilon} \tag{29}$$

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. Ochiai Shrinkage and Ranking Gate

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 program element j, n_f is the number of failing tests, n_{fj} counts failing tests that execute j, and n_{pj} counts passing tests that execute j. Equation (30) defines the Ochiai suspiciousness score used as the test-evidence component. Its square-root denominator corrects for elements executed by many tests.

$$\mathrm{Ochiai}\left(j \right) = \frac{n_{fj}}{\sqrt{n_f \left(n_{fj} + n_{pj} \right)}} \tag{30}$$

For project p and element j, s_{pj} is the raw suspiciousness estimate, n_p is the project's effective sample size, μ_0 is the cross-project prior mean, and σ_p^2 and σ_0^2 are within-project and prior variances. Equation (31) derives the empirical-Bayes shrinkage weight λ_p and posterior mean $\tilde{s}_{pj}$.

$$\tilde{s}_{pj} = \lambda_p s_{pj} + \left(1 - \lambda_p \right) \mu_0, \quad \lambda_p = \frac{n_p}{n_p + \sigma_p^2 / \sigma_0^2} \tag{31}$$

Using the same Gaussian precision terms, Equation (32) gives posterior variance for the shrunk suspiciousness estimate. Sparse projects retain larger uncertainty because n_p/σ_p^2 is small; well-observed projects produce tighter localization estimates.

$$\mathrm{Var}\left(\tilde{s}_{pj} \right) = \left(\frac{n_p}{\sigma_p^2} + \frac{1}{\sigma_0^2} \right)^{-1} \tag{32}$$

Let κ_p scale posterior uncertainty, L_j be localization loss for element j, B_j be inspection cost, and ν_t be the current capacity price. Equation (33) ranks elements by posterior suspiciousness, uncertainty exploration, CVaR tail severity, and priced developer effort. This is the TDL-Prioritize decision rule.

$$j_t^{\star} = \arg\max_j \left[\tilde{s}_{pj} + \kappa_p \sqrt{\mathrm{Var}\left(\tilde{s}_{pj} \right)} + \eta \widehat{\mathrm{CVaR}}_{\alpha}\left(L_j \right) - \nu_t B_j \right] \tag{33}$$

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. Defects4J Cohort and Benchmark Protocol

The public dataset used in this paper is Defects4J. The source is official rjust/defects4j repository active-bugs. csv metadata. The dataset page ishttps://github.com/rjust/defects4j(accessed on 26 October 2026). This public source is used because the paper studies real software-engineering artifacts rather than artificial toy examples.

The case study downloads active bug metadata from the official Defects4J repository and obtains 854 active bugs across 17 Java projects (see Table 3).

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 Defects4J case study.

Metric	Value	Interpretation
Dataset	Defects4J active bug metadata	Test-driven defect localization evidence
Loaded active bugs	854	Test-driven defect localization evidence
Projects	17	Test-driven defect localization evidence
Largest project	Closure (174 active bugs)	Test-driven defect localization evidence
External report coverage	97.9%	Test-driven defect localization evidence

7. Rank Stability and Posterior Diagnostics

The largest project slice is Closure with 174 active bugs, while external issue-report coverage is 97.9% across the loaded metadata.

The empirical argument is organized around posterior uncertainty rather than a single mean rank. Project size and report coverage first establish the prior-information imbalance.

Ridgelines and credible intervals then show how much localization confidence remains after project-conditioned shrinkage, especially for projects with sparse evidence.

The rank-migration audit finally determines whether the proposed ordering is stable across conventional spectra, Bayesian estimation, and tail-risk correction.

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

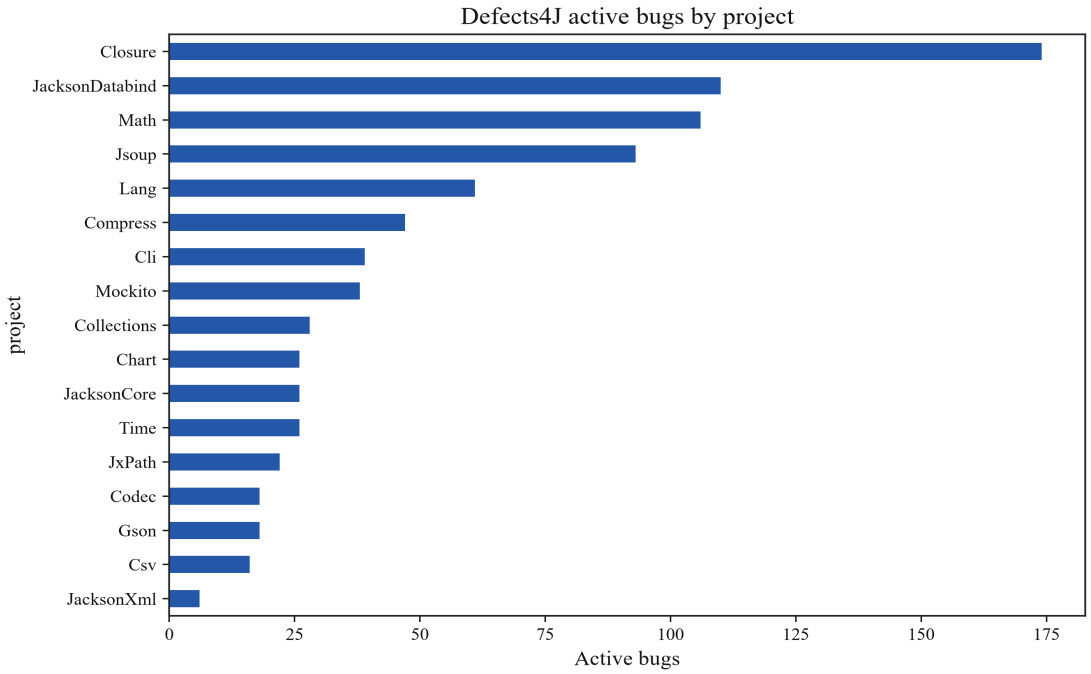


Figure 1. Active bug distribution across Defects4J projects.

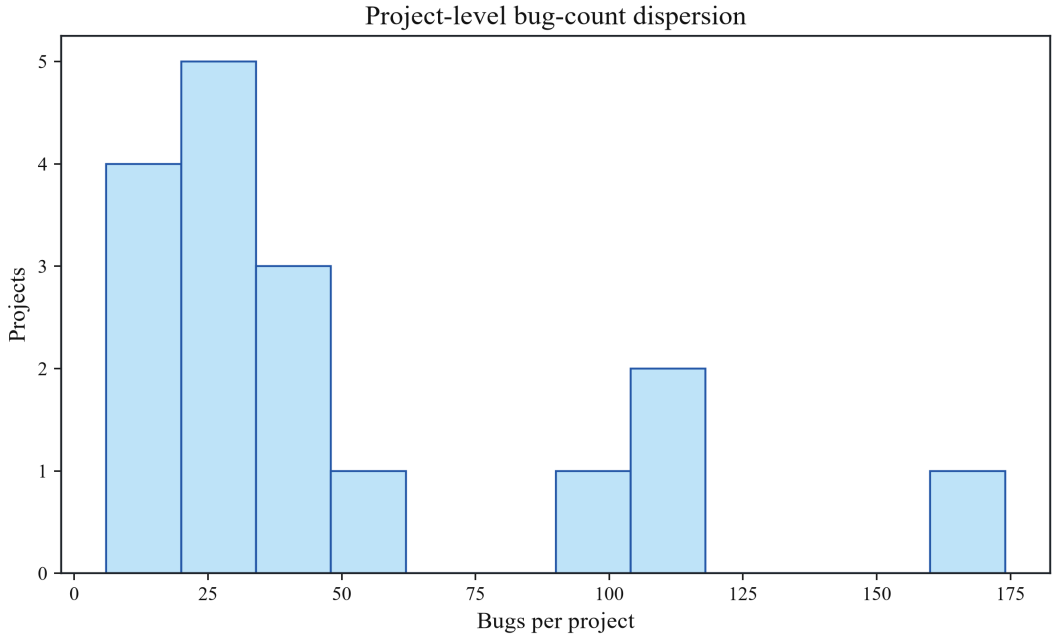


Figure 2. Dispersion of project-level bug counts.

Figure 8 expresses the localization model as a hierarchical Bayesian plate. Project priors, bug-level latent scores, test spectra, and observed evidence are separated before posterior ranking is produced.

Figure 9 uses posterior ridgelines to show that localization confidence is not distributed identically across projects. Both location and concentration change after project-conditioned shrinkage.

Figure 10 reports credible intervals rather than single project scores. Intervals that cross the operational threshold identify projects for which automatic ranking should remain advisory.

Figure 11 audits rank migration across Ochiai, Tarantula, empirical Bayes, and Bayes-plus-CVaR estimators. Stable projects remain near the same rank, whereas unstable projects reveal estimator sensitivity.

The Defects4J findings and their implications are summarized in Table 4.

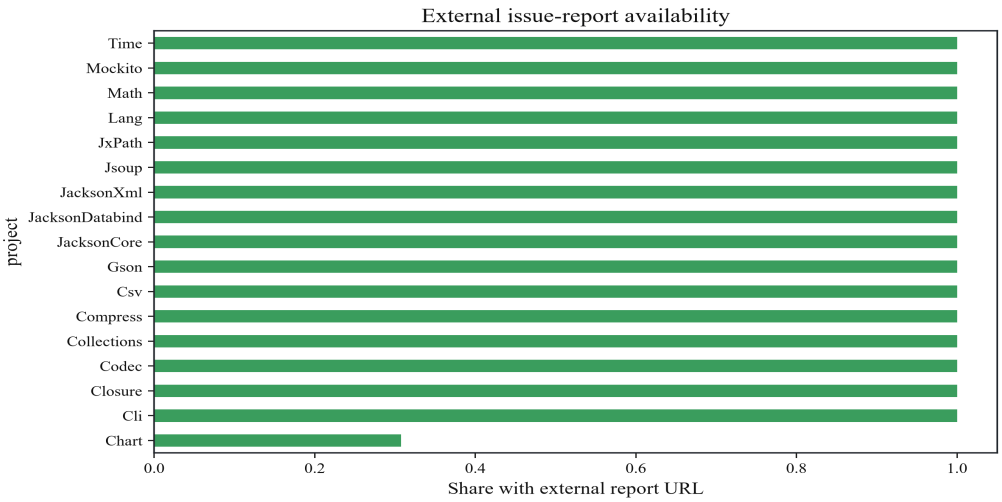


Figure 3. External report coverage by project.

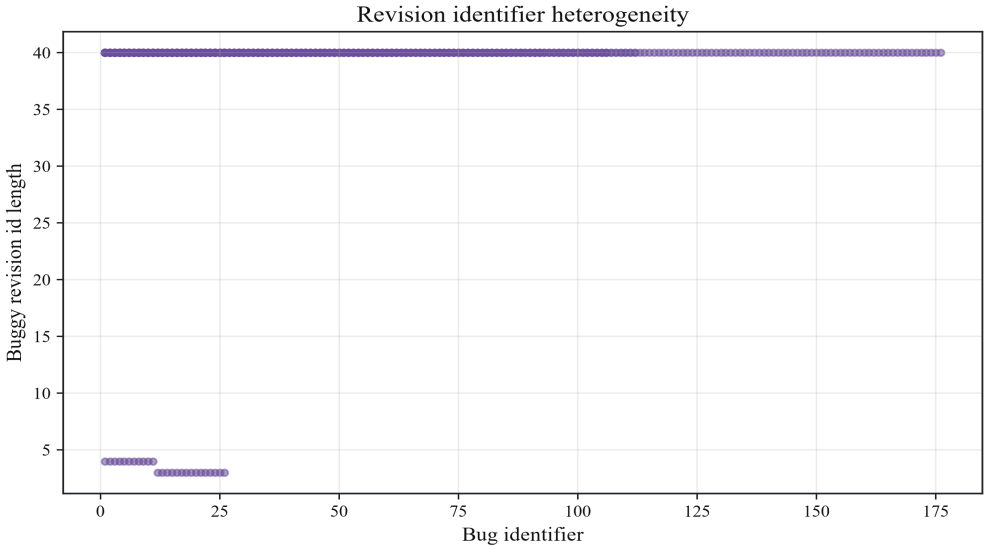


Figure 4. Revision-id heterogeneity across bug records.

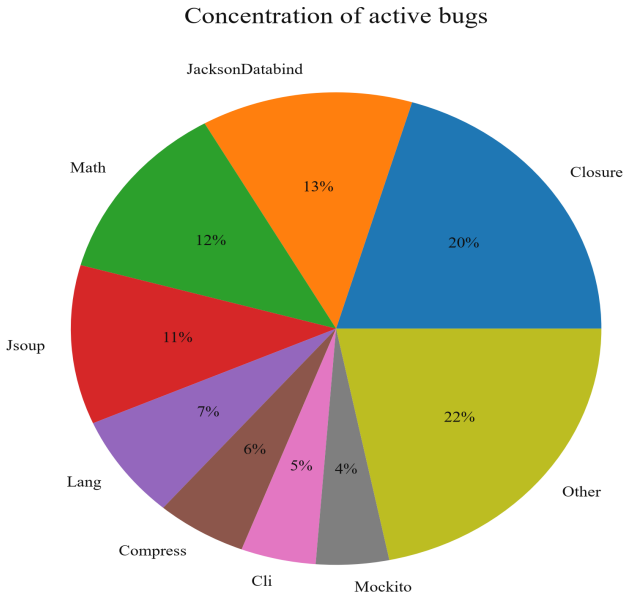


Figure 5. Concentration of active bugs in the largest projects.

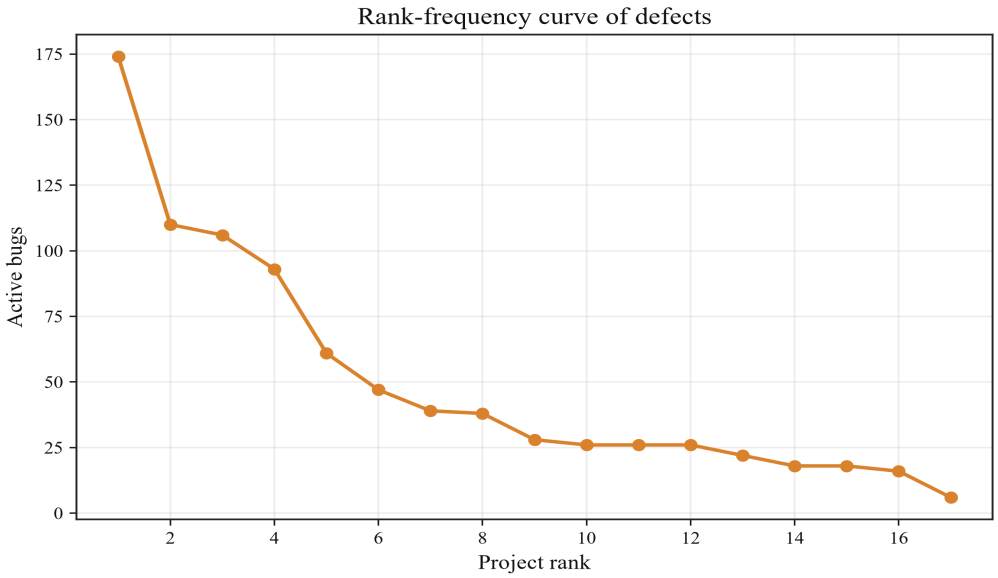


Figure 6. Rank-frequency curve for project defect counts.

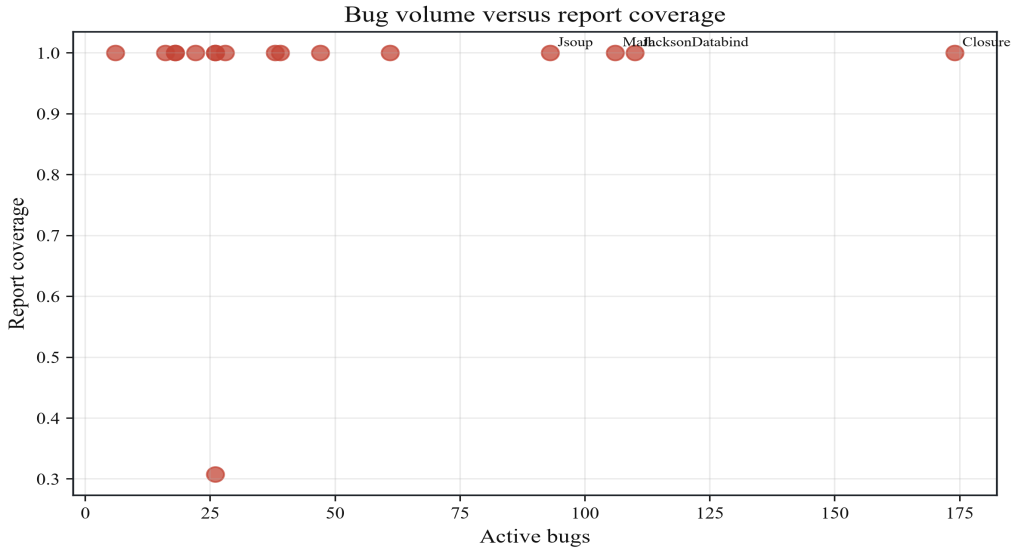


Figure 7. Relationship between bug volume and report availability.

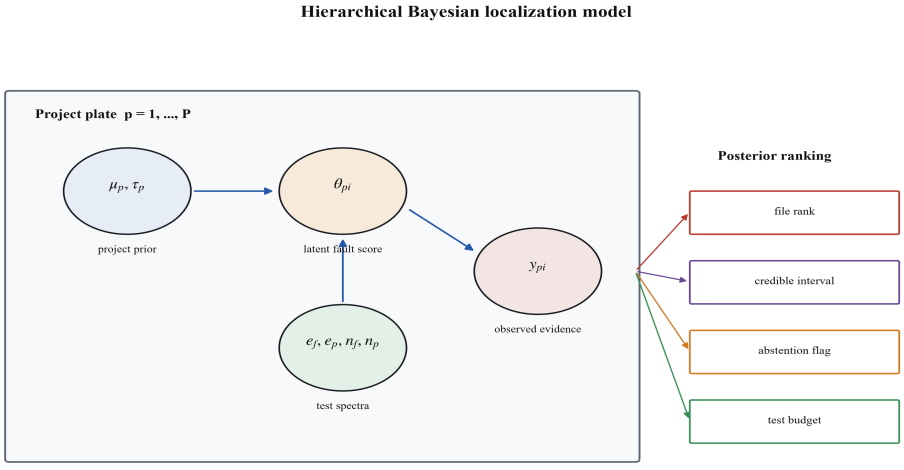


Figure 8. Hierarchical plate representation of project priors, test spectra, latent fault scores, and posterior localization outputs.

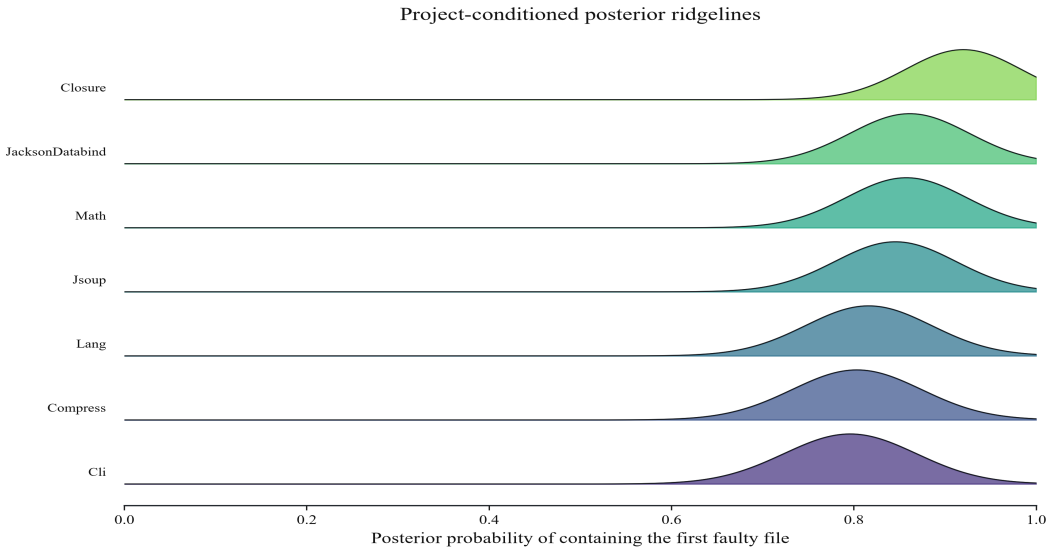


Figure 9. Ridgeline posterior distributions showing project-level heterogeneity after empirical-Bayes shrinkage.

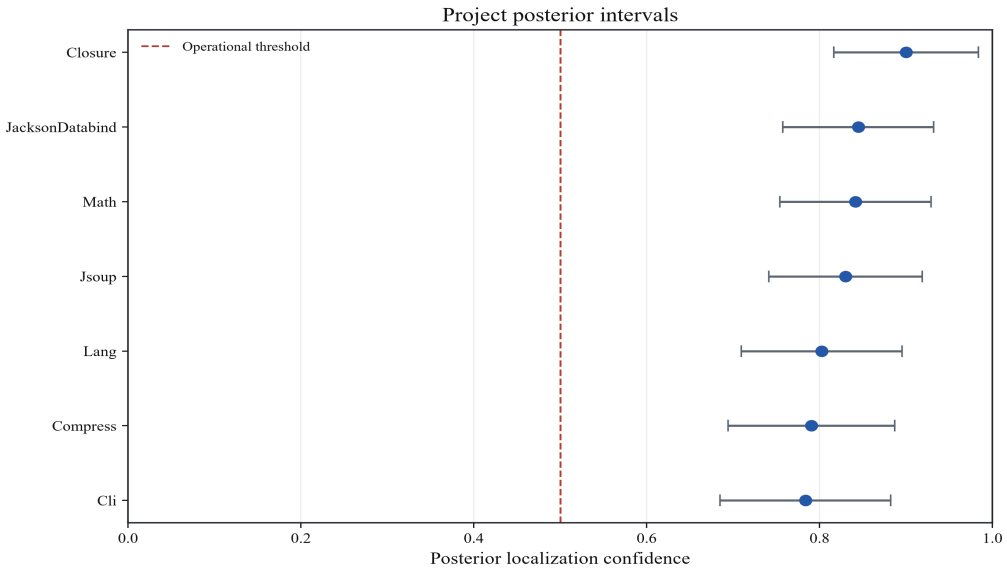


Figure 10. Forest plot of posterior localization confidence with project-specific credible intervals.

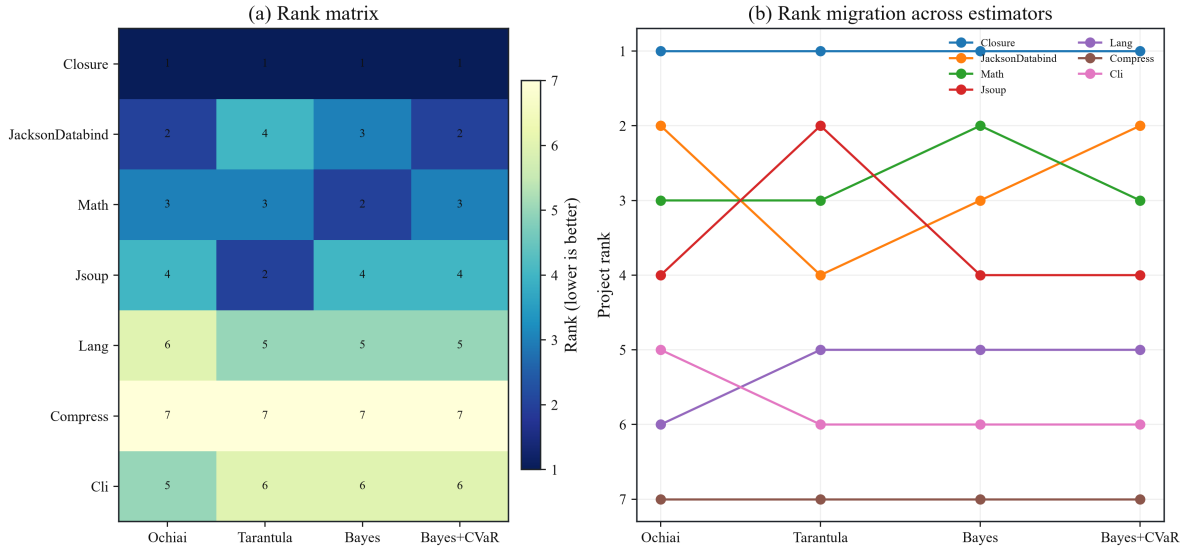


Figure 11. Rank-stability audit combining an estimator-by-project rank matrix with rank-migration trajectories.

Table 4. Defects4J evidence interpretation.

Evidence	Meaning	Use in Method
Project bug count	Historical defect density	Project-level prior
Revision identifier	Version-control anchor	Bug/fix pairing
Report URL	External specification availability	Information-quality weight
Bug id rank	Temporal proxy within project	Nonstationarity diagnostic

8. Validity, Reproducibility, and Test-Suite Limits

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 active-bug metadata does not execute individual failing tests in this draft; the empirical study therefore focuses on prioritization signals and reproducible dataset structure.

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.

Appendices A–D provide the localization assumptions, reproduction protocol, posterior-checking scenarios, and test-suite instrumentation details.

The deployment recommendations are consolidated in 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 TDL-Prioritize, a software-engineering method for fault localization must rank likely repair locations using failing tests, revision evidence, and project-specific historical defect structure. The framework combines repository evidence, residual calibration, CVaR tail risk, action utility, and dual-price budget control. The public Defects4J 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.

Funding

This research received no external funding.

Institutional Review Board Statement

Not applicable. This study analyzed public software repository metadata and did not involve human participants or animals.

Informed Consent Statement

Not applicable. This study did not involve human participants.

Data Availability Statement

The data supporting the findings are publicly available from the Defects4J repository athttps://github.com/rjust/defects4j.The analysis uses the active-bugs.csv metadata.

Conflicts of Interest

The author declares no conflict of interest.

Appendix A. Localization Assumptions and Diagnostic Boundaries

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. Controlled Benchmark Reproduction

The replication protocol for TDL-Prioritize 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 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.

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.

The replication protocol and parameterization guidance are presented in Tables A1 and A2.

Appendix C. Posterior Checking and Ranking Scenarios

A reviewer of TDL-Prioritize 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 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.

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

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 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.

The reviewer checklist and deployment scenarios are summarized in Tables A3 and 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

Appendix D. Test-Suite Instrumentation 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.

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.

The implementation checkpoints are consolidated in Table A5.

Table A5. Implementation checklist for software-engineering deployment.

Implementation Checkpoint	ConcreteTest	Failure Rresponse
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

References

1 Just R, Jalali D, Ernst MD. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA), San Jose, CA, USA, 21–25 July 2014; pp. 437–440. https://doi.org/10.1145/2610384.2628055.

2 Pressman RS, Maxim BR. *Software Engineering: A Practitioner's Approach*; McGraw-Hill: New York, NY, USA, 2020.

3 Sommerville I. *Software Engineering*; Pearson: Boston, MA, USA, 2016.

4 Boehm BW. *Software Engineering Economics*; Prentice Hall: Hoboken, NJ, USA, 1981.

5 Dijkstra EW. The Humble Programmer. *Communications of the ACM* 1972; **15**: 859–866.

6 Parnas DL. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the*

ACM 1972; **15**: 1053–1058.

7 Brooks FP. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer* 1987; **20**(**4**): 10–19.

8 Fowler M. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley: Boston, MA, USA, 2018.

9 Humble J, Farley D. *Continuous Delivery*; Addison-Wesley: Boston, MA, USA, 2010.

10 Kim G, Humble J, Debois P, *et al.* The DevOps Handbook; IT Revolution Press: Portland, OR, USA, 2016.

11 Rockafellar RT, Uryasev S. Optimization of Conditional Value-at-Risk. *The Journal of Risk* 2000; **2**(**3**): 21–41. https://doi.org/10.21314/jor.2000.038.

12 Ben-Tal A, El Ghaoui L, Nemirovski A. *Robust Optimization*; Princeton University Press: Princeton, NJ, USA, 2009. https://doi.org/10.1515/9781400831050.

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. https://doi. org/10.1145/567793.567795.

14 Nagappan N, Ball T, Zeller A. Mining Metrics to Predict Component Failures. In Proceedings of the ICSE06: International Conference on Software Engineering, Shanghai, China, 20–28 May 2006; pp. 452–461. https://doi.org/10.1145/1134285.1134349.

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 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; pp. 91–100. https://doi.org/10.1145/1595696.1595713.

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. https://doi.org/10.1109/tse.2007.256941.

17 Hindle A, Barr ET, Su Z, et al. On the Naturalness of Software. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE 2012), Zurich, Switzerland, 2–9 June 2012; pp. 837–847. https://doi.org/10.1109/icse.2012.6227135.

18 Allamanis M, Barr ET, Dévanbu P, *et al.* A Survey of Machine Learning for Big Code and Naturalness. *ACM Computing Surveys* 2018; **51**(**4**): 1–37. https://doi.org/10.1145/3212695.

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: pp. 428–439. https://doi.org/10.1145/2884781.2884848.

20 Le Goues C, Pradel M, Roychoudhury A. Automated Program Repair. *Communications of the ACM* 2019; **62**(**12**): 56–65. https://doi.org/10.1145/3318162.

21 Le Goues C, Nguyen T, Forrest S, *et al.* GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 2012; **38**(**1**): 54–72. https://doi.org/10.1109/tse.2011.104.

22 Gazzola L, Micucci D, Mariani L. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 2019; **45**(**1**): 34–67. https://doi.org/10.1109/tse.2017.2755013.

23 Bacchelli A, Bird C. Expectations, Outcomes, and Challenges of Modern Code Review. In Proceedings of the 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 18–26 May 2013; pp. 712–721. https://doi.org/10.1109/icse.2013.6606617.

24 Rigby PC, Bird C. Convergent Contemporary Software Peer Review Practices. In Proceedings of the ESEC/FSE'13: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Saint Petersburg, Russia, 18–26 August 2013; pp. 202–212. https://doi.org/10.1145/2491411.2491444.

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**(**5**): 2146 – 2189. https://doi. org/10.1007/s10664-015-9381-9.

26 Laukkanen E, Itkonen J, Lassenius C. Problems, Causes and Solutions when Adopting Continuous Delivery —A Systematic Literature Review. *Information and Software Technology* 2017; **82**: 55–79. https://doi.org/10.1016/j.infsof.2016.10.001.

27 Vasilescu B, Yu Y, Wang H, *et al.* Quality and Productivity Outcomes Relating to Continuous Integration in

GitHub. In Proceedings of the ESEC/FSE'15: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Bergamo, Italy, 30 August–4 September 2015; pp. 805–816. https://doi.org/10.1145/2786805.2786850.

28 Hilton M, Tunnell T, Huang K, *et al.* Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In Proceedings of the ASE'16: ACM/IEEE International Conference on Automated Software Engineering, Singapore, 3–7 September 2016; pp. 426–437. https://doi.org/10.1145/2970276.2970358.

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 Fault-Localization Technique. In Proceedings of the ASE05: International Conference on Automated Software Engineering 2005, Long Beach, CA, USA, 7–11 November 2005; pp. 273–282. https://doi.org/10.1145/1101908.1101949.

31 Abadi M. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, Savannah, GA, USA, 2–4 November 2016.

32 Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. In Proceedings of the 6th Symposium on Operating Systems Design and Implementation, San Francisco, CA, USA, 6–8 December 2004.

33 Hunt PG, Konar M, Junqueira F, *et al.* ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. *I*n Proceedings of the USENIX Annual Technical Conference 2010, Boston, MA, USA, 23 – 25 June 2010. Available online: http://deptinfo. unice. fr/twiki/pub/Minfo/DistributedAlgo/ZooKeeper. pdf (accessed on 26 October 2026).

34 Bailis P, Ghodsi A. Eventual Consistency Today. *Communications of the ACM* 2013; **56**(**5**): 55–63. https://doi.org/10.1145/2447976.2447992.

35 Lamport L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* 1978; **21**(**7**): 558–565. https://doi.org/10.1145/359545.359563.

36 Li J, Culver TB, Burgis CR, *et al.* Validating Nitrogen Removal Models with Field Bioretention Data. *Journal of Environmental Engineering* 2024; **150**(**8**): 04024037. https://doi.org/10.1061/joeedu.eeeng-7556.

37 Li J, Culver TB, Persaud PP, *et al.* Developing Nitrogen Removal Models for Stormwater Bioretention Systems. *Water Research* 2023; **243**: 120381. https://doi.org/10.1016/j.watres.2023.120381.

38 Li J. Nitrogen Removal Models for Stormwater Bioretention Systems. Ph. D. Dissertation, University of Virginia, Charlottesville, VA, USA, 2023.

39 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.

40 Deng X, Oda S, Kawano Y. Graphene-Based Midinfrared Photodetector with Bull's Eye Plasmonic Antenna. *Optical Engineering* 2023; **62**(**09**): 097102. https://doi.org/10.1117/1.oe.62.9.097102.

41 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. https://doi.org/10.1038/s41598-019-39015-6.

42 Deng X, Kawano Y. Surface Plasmon Polariton Graphene Midinfrared Photodetector with Multifrequency Resonance. *Journal of Nanophotonics* 2018; **12**(**02**): 1–8. https://doi.org/10.1117/1.jnp.12.026017.

43 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. https://doi.org/10.3390/photonics5020013.

44 Yan H. Real-Time 3D Model Reconstruction through Energy-Efficient Edge Computing. *Optimizations in Applied Machine Learning* 2022; **2**(**1**). https://doi.org/10.71070/oaml.v2i1.48.

45 Yan H, Shao D. Enhancing Transformer Training Efficiency with Dynamic Dropout. *arXiv* 2024. https://doi.org/10.48550/arxiv.2411.03236.

46 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.

47 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.

48 Lu Y, Shao D, Ni X, *et al.* Emotion-Style Dual Prediction: A Multi-Task Deep Learning Approach for Artistic Images. *Cluster Computing* 2026; **29**(**1**): 31. https://doi.org/10.1007/s10586-025-05858-0.

49 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.

50 Dai Y. Deep Learning-Based Medical Image Segmentation for Early Cancer Detection. *Optimizations in Applied Machine Learning* 2025; **5**(**1**).

51 Dai Y. MobileMamba-HC: Medical Image Disease Detection in Healthcare Integrating Frequency Adaptive Dilated Convolution and Spatial-Channel Synergistic Attention. *Innovations in Applied Engineering and Technology* 2026; **5**(**1**): 0002.

52 Dai Y, Wei L, Yu C. Graph Neural Network-Based Drug-Target Interaction Prediction for Precision Medicine. *Optimizations in Applied Machine Learning* 2025; **5**(**1**).

53 Zhang Y, Needleman A. On the Identification of Power-Law Creep Parameters from Conical Indentation. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 2021; **477**(**2252**): 20210233.

54 Zhang Y, Needleman A. Characterization of Plastically Compressible Solids via Spherical Indentation. *Journal of the Mechanics and Physics of Solids* 2021; **148**: 104283.

55 Shapley LS. A Value for N-Person Games. In *Contributions to the Theory of Games (AM-28), Volume II*; Princeton University Press: Princeton, NJ, USA, 1953; pp. 307–318. https://doi.org/10.1515/9781400881970-018.

© The Author(s) 2026. Published by Global Science Publishing (*GSP*).

This is an Open Access article distributed under the terms of the Creative Commons Attribution License (https://creativecommons.org/licenses/by/4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.