

From Change Risk to Adaptive Validation: An Empirical Framework for Risk-Aware Software Life-Cycle Management

Amit Kumar Singh

School of Engineering and Technology
K.R. Mangalam University
Gurugram, Haryana, India

Received: Jul 26, 2026

Accepted: Aug 23, 2026

Published: Oct 06, 2026

Abstract

Continuous integration and delivery pipelines repeatedly allocate build, test, analysis, and review effort to software changes whose failure risk is heterogeneous. This paper presents RA-SLCM, a risk-aware software life-cycle management framework in which commit-time evidence is converted into a calibrated prioritization signal for adaptive CI/CD validation. The empirical study uses the public ApacheJIT corpus and a chronological protocol that prevents future-to-past leakage: 30,218 commits from 2003-2014 are used for model construction and 14,616 commits from 2015-2016 form an untouched future holdout. Twelve process and change metrics available at commit time are evaluated with class-weighted logistic regression and random forest models. On the temporal holdout, the random forest achieves ROC-AUC 0.819 (95% bootstrap CI 0.813-0.826), average precision 0.813, precision 0.728, recall 0.764, and F1 0.746. Training-derived risk thresholds produce clearly separated future strata: observed defect-inducing rates are 24.4%, 65.4%, and 89.3% for low-, moderate-, and high-risk commits. The highest-risk 20% of commits contain 35.5% of all defect-inducing commits, a 1.78 lift over random inspection. Unlike studies that optimize prediction alone, RA-SLCM explicitly positions change-risk estimation as a control signal for downstream validation depth while separating measured predictive evidence from unmeasured CI/CD runtime effects. The results establish a reproducible empirical basis for adaptive validation and identify the deployment measurements required for a subsequent end-to-end CI/CD study.

General Terms: Software Engineering, Experimentation, Software Quality

Keywords: Software life-cycle management, DevOps, CI/CD, just-in-time defect prediction, risk-aware testing, software quality assurance

1 Introduction

Modern software life-cycle management (SLCM) is increasingly organized around short iterations, continuous integration (CI), automated testing, and continuous delivery/deployment (CD). Every accepted change can trigger compilation, unit and integration tests, static analysis, security checks, packaging, and deployment gates. A fixed pipeline is operationally simple, but it treats heterogeneous changes as if they carried identical failure risk.

Change-level defect prediction offers a complementary view. Just-in-time defect prediction (JIT-DP) estimates whether a commit is likely to be defect-inducing when it enters the repository [2]. Recent work has improved semantic representations, transfer learning, online learning, and explainability [4, 9, 10, 12, 8]. Yet a practical SLCM question remains: how should a prediction be translated into a validation decision?

We address this gap with Risk-Aware Software Life-Cycle Management (RA-SLCM). The novelty is not a claim that random forests outperform the latest JIT-DP architectures. Instead, the contribution is an empirically evaluated bridge from commit-time risk estimation to *adaptive validation control*. The estimator and validation policy are deliberately separated so that an organization can alter test depth, review requirements, or security gates without changing the underlying risk model.

The study asks three research questions: **RQ1:** Can commit-time evidence discriminate defect-inducing future changes under chronological evaluation? **RQ2:** Do training-derived risk thresholds produce operationally distinct future risk strata? **RQ3:** How much defect-inducing change concentration is obtained when validation effort is focused on the highest-risk commits?

The contributions are: (1) a risk-to-validation SLCM architecture; (2) a leakage-resistant temporal evaluation on 44,834 commits; (3) interpretable logistic and nonlin-

ear random-forest baselines with uncertainty reporting; (4) risk-tier and effort-aware analyses that directly support differentiated validation; and (5) an explicit boundary between measured prediction results and CI/CD effects that still require an instrumented deployment.

2 Related Work and Positioning

JIT quality assurance was established as a change-level alternative to release-level prediction, with process metrics such as diffusion, size, history, and developer experience providing useful signals [2]. Effort-aware research subsequently emphasized that prediction should be evaluated in relation to the amount of code or review effort consumed, not accuracy alone [3, 6].

Recent studies increasingly exploit richer representations. JIT-Fine combines semantic and expert features [4]; JIT-BiCC learns bi-modal representations from code changes and commit messages [9]; and JIT-CF combines contrastive learning with feature fusion [11]. Other work addresses cross-project transfer [10], online and human-in-the-loop operation [12], and concept drift and imbalance [13]. Explainability has also become central: FoX provides formal explanations for JIT predictions [8]. Evaluation methodology itself is nontrivial because delayed labels and retrospective protocols can bias performance estimates [7].

Table 1 positions RA-SLCM relative to representative work. The present paper intentionally uses conventional models: its contribution lies in temporal validation, operational stratification, and the mapping of risk to life-cycle validation policy rather than a new representation learner.

3 RA-SLCM Framework

For a change c_i , let x_i be its observable repository evidence and $p_i = P(y_i = 1 | x_i)$ the estimated probability that the change is defect-inducing. RA-SLCM maps this score to a validation action:

$$V(c_i) = g(p_i, \tau_1, \tau_2), \quad (1)$$

where τ_1 and τ_2 partition commits into low-, moderate-, and high-risk strata. A deployment can associate these strata with progressively stronger controls, for example: (i) build, static checks, and smoke tests; (ii) unit plus integration testing; and (iii) full regression, security analysis, and/or manual approval. These actions are policy examples, not measured outcomes in the present experiment.

The operational loop is: Commit $\rightarrow$ evidence extraction $\rightarrow$ risk score $\rightarrow$ validation tier $\rightarrow$ test/deployment

outcome $\rightarrow$ historical feedback. This architecture makes the risk estimator replaceable and allows later calibration against actual pipeline cost and escaped defects.

4 Experimental Methodology

4.1 Dataset and Label Construction

ApacheJIT is a public JIT-DP corpus containing 106,674 commits from popular Apache projects, including 28,239 bug-inducing changes [5]. Its construction links fixed bug reports to fixing commits and then uses a filtered SZZ-based procedure to identify candidate bug-inducing changes. The archive supplies commit identifiers, dates, process/change metrics, and a buggy label. Because SZZ-derived labels can contain false positives and false negatives, we treat the labels as mined proxies for defect induction rather than ground truth and return to this limitation in Section 7.

The experiment uses the released balanced training corpus and imposes a further chronological split: commits dated 2003–2014 are used for model construction and commits dated 2015–2016 form an untouched future holdout. This yields 30,218 training commits and 14,616 holdout commits; 7,255 holdout commits (49.64%) are labelled defect-inducing. No future labels participate in preprocessing, threshold selection, or fitting.

4.2 Features and Preprocessing

Twelve metrics available at commit time are retained: lines added (LA), lines deleted (LD), files touched (NF), directories (ND), subsystems (NS), change entropy (ENT), number of developers touching the files (NDEV), average age of changed files (AGE), unique prior changes (NUC), author experience (AEXP), recent author experience (AREXP), and subsystem experience (ASEXP). These follow the process-metric family used in established JIT studies [2, 5]. Non-negative variables are transformed using $\log(1 + x)$. Logistic regression additionally uses robust scaling fitted only on the training period.

4.3 Models, Thresholds, and Metrics

We evaluate class-weighted logistic regression as an interpretable linear baseline and a 300-tree random forest with minimum leaf size five and balanced-subsample weighting. The classification threshold is selected exclusively from training-period predictions by maximizing F1 over $[0.05, 0.95]$ in increments of 0.005. Final models are then evaluated once on the future holdout.

Table 1: Positioning against representative recent JIT defect-prediction research.

Study	Primary focus	Evaluation emphasis	Operational action	Distinction from RA-SLCM
Ni et al. [4]	Semantic + expert features	Prediction and localization	Defect localization	Representation-centric
Jiang et al. [9]	Bi-modal change representation	F1/AUC	Risk prediction	Pre-trained semantic model
Yu et al. [8]	Formal explainability	Correct/actionable explanations	Explain/mitigate risk	Explanation-centric
Liu et al. [12]	Human-in-the-loop online JIT	Online effectiveness	SQA feedback	Online learning and feedback
JIT-CF [11]	Contrastive feature fusion	Predictive performance	Risk prediction	Representation-centric
RA-SLCM	Risk-to-validation control	Temporal hold-out, tiers, prioritization	Adaptive CI/CD validation depth	Separates estimator from policy

We report ROC-AUC, average precision (AP), precision, recall, and F1. Because a point estimate can conceal sampling variability, the random-forest ROC-AUC is accompanied by a 95% nonparametric bootstrap confidence interval from 1,000 resamples. For prioritization, holdout commits are ranked by predicted risk and defect-inducing recall is measured in the highest-risk 20%.

4.4 Risk-Tier Protocol

The 50th and 80th percentiles of *training* risk scores define τ_1 and τ_2 . These thresholds are frozen before evaluation and then applied to future commits. This avoids choosing tier boundaries after observing holdout outcomes. We report the number of commits, defect-inducing commits, and observed defect rate in each tier.

5 Results

5.1 RQ1: Future-Commit Discrimination

Table 2 reports temporal-holdout performance. Logistic regression obtains AUC 0.797 and F1 0.740. The random forest improves discrimination to AUC 0.819 and AP 0.813; its bootstrap AUC interval is 0.813 0.826. At the training-selected threshold 0.490, precision is 0.728, recall 0.764, and F1 0.746. Thus, the evidence supports **RQ1**: conventional commit-time metrics retain useful predictive information on chronologically later changes.

5.2 RQ2: Operational Risk Stratification

Frozen training-derived thresholds assign 6,990 holdout commits to low risk, 5,279 to moderate risk, and 2,347 to

Table 2: Performance on the 2015 2016 temporal holdout.

Model	AUC	AP	Prec.	Rec.	F1
Logistic	.797	.783	.655	.852	.740
Random forest	.819	.813	.728	.764	.746

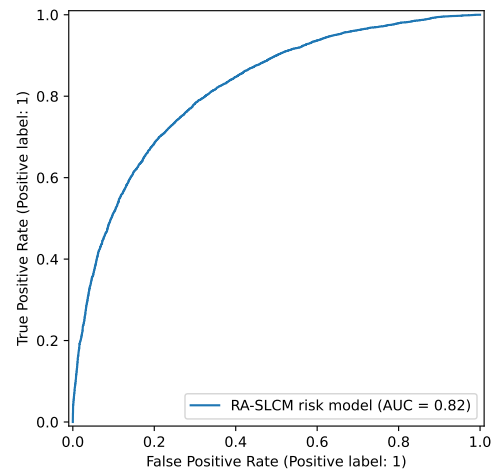


Figure 1: ROC curve of the selected risk model on future commits.

high risk. The corresponding observed defect-inducing rates are 24.4%, 65.4%, and 89.3% (Table 3). High-risk commits are therefore about 3.66 times as defect-dense as low-risk commits. The monotonic separation answers **RQ2** positively and provides a directly interpretable basis for differentiated validation.

Table 3: Future outcomes under training-derived risk tiers.

Tier	Commits	Buggy	Rate
Low	6,990	1,707	24.4%
Moderate	5,279	3,452	65.4%
High	2,347	2,096	89.3%

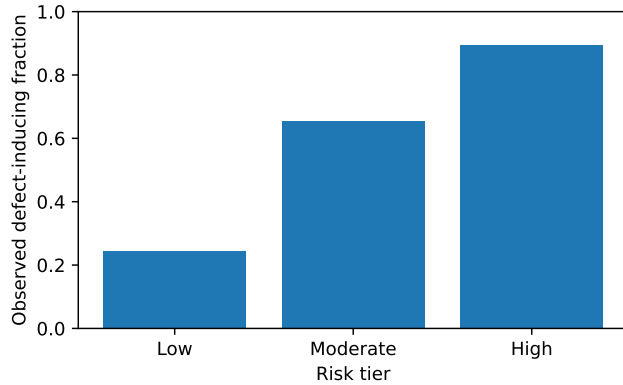


Figure 2: Observed defect-inducing fraction by RA-SLCM risk tier.

5.3 RQ3: Effort-Aware Prioritization

The highest-risk 20% of holdout commits contain 35.5% of all defect-inducing commits. Under random inspection, expected coverage is 20%, giving a lift of $35.5/20 = 1.78$. This answers **RQ3**: concentrating additional validation on the risk-ranked head of the stream exposes substantially more defect-inducing changes per inspected commit than random allocation. This is a ranking result; it is not a claim about the sensitivity of any particular test suite.

5.4 Feature Sensitivity

Permutation analysis (Fig. 3) indicates that predictive information is distributed across size, diffusion, history, and experience variables. This is consistent with the SLCM premise that risk should be inferred from multiple life-cycle signals rather than a single threshold such as lines changed.

6 Discussion

The experiment establishes the prerequisite for risk-aware validation: a score derived before release can stratify future commits into groups with sharply different observed defect rates. This allows a CI/CD controller to allocate validation effort according to risk while preserving a minimum mandatory test floor for every change. The latter

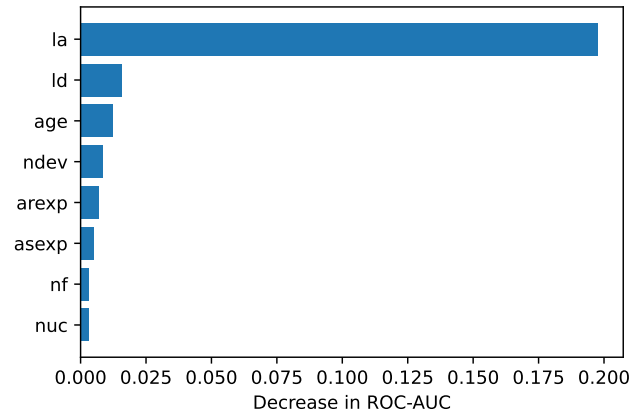


Figure 3: Permutation importance measured as decrease in ROC-AUC.

is important because the low-risk tier still contains 1,707 defect-inducing commits; RA-SLCM therefore supports *differentiation*, not test bypass.

The results should also be interpreted relative to recent JIT-DP research. State-of-the-art semantic models may achieve stronger prediction on their target benchmarks [9, 11]. RA-SLCM is complementary: any stronger estimator can replace the random forest if it produces a reliable commit-time score. The framework contribution is the explicit interface between prediction and life-cycle control, together with an evaluation that asks whether fixed training-derived tiers remain meaningful on future data.

No Jenkins wall-clock time, compute consumption, test-suite detection rate, or production escaped-defect rate was measured in this study. Accordingly, the paper makes no percentage claim about CI/CD time or cost savings. A valid end-to-end claim requires replaying commits through instrumented pipelines under fixed and adaptive policies and comparing both resource consumption and quality outcomes.

7 Threats to Validity

Construct validity. ApacheJIT uses repository mining and a filtered SZZ procedure to infer bug-inducing commits [5]. SZZ-family labels can be noisy, and recent work continues to investigate label reliability. The reported “buggy” outcome should therefore be understood as the dataset’s defect-inducing label, not a manually verified defect for every commit.

Internal validity. Hyperparameters, preprocessing, and threshold search influence performance. We reduce leakage by fitting transformations and thresholds only on

the training period and evaluating the future holdout once. Nevertheless, the current experiment does not include nested hyperparameter optimization or repeated project-wise temporal folds.

External validity. Apache projects may not represent proprietary repositories, mobile applications, safety-critical software, or organizations with different review practices. Cross-project and online studies show that distribution shift remains important [10, 12, 13].

Evaluation validity. Delayed labels and retrospective construction can distort JIT evaluation [7]. Our chronological holdout reduces future information leakage but does not reproduce a fully online delayed-label environment. Furthermore, ranking defect-inducing commits is not equivalent to proving that a particular CI test will detect the defect.

8 Reproducibility and Artifact Scope

The analysis uses the public ApacheJIT archive and a deterministic date-based split. Random seeds are fixed to 42 for the random forest and bootstrap procedure. The accompanying manuscript bundle contains the reported model summary and generated figures. The original ApacheJIT dataset is not redistributed; it is available from the cited archival record [5]. For a fully auditable submission, the analysis script, package versions, and checksum of the downloaded dataset should be archived with the authors' replication package before publication.

9 Conclusion

RA-SLCM reframes commit-level defect risk as an input to software life-cycle control rather than an isolated prediction task. Across 14,616 chronologically future commits, the selected random forest achieves ROC-AUC 0.819 (95% CI 0.813 0.826), and frozen risk thresholds yield defect-inducing rates of 24.4%, 65.4%, and 89.3% across low, moderate, and high tiers. The highest-risk 20% of commits contain 35.5% of all defect-inducing changes. These findings provide an empirical basis for adaptive validation while avoiding unsupported claims about pipeline runtime or defect removal. The next step is an instrumented CI/CD replay study that jointly measures validation cost, release latency, test outcomes, and escaped defects under fixed and risk-aware policies.

References

- [1] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.
- [2] Y. Kamei et al., "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757 773, 2013.
- [3] Y. Yang et al., "Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models," in *Proc. ESEC/FSE*, 2016.
- [4] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, and D. Lo, "The best of both worlds: Integrating semantic features with expert features for defect prediction and localization," in *Proc. ESEC/FSE*, pp. 672 683, 2022.
- [5] H. Keshavarz and M. Nagappan, "ApacheJIT: A large dataset for just-in-time defect prediction," in *Proc. 19th Int. Conf. Mining Software Repositories*, 2022, doi:10.1145/3524842.3527996. Dataset: doi:10.5281/zenodo.5907002.
- [6] J. Çarka, M. Esposito, and D. Falessi, "On effort-aware metrics for defect prediction," *Empirical Software Engineering*, vol. 27, art. 152, 2022, doi:10.1007/s10664-022-10186-7.
- [7] L. Song, L. L. Minku, and X. Yao, "On the validity of retrospective predictive performance evaluation procedures in just-in-time software defect prediction," *Empirical Software Engineering*, vol. 28, art. 124, 2023, doi:10.1007/s10664-023-10341-8.
- [8] J. Yu, M. Fu, A. Ignatiev, C. Tantithamthavorn, and P. Stuckey, "A formal explainer for just-in-time defect predictions," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, art. 187, 2024, doi:10.1145/3664809.
- [9] Y. Jiang, B. Shen, and X. Gu, "Just-in-time software defect prediction via bi-modal change representation learning," *Journal of Systems and Software*, vol. 219, art. 112253, 2025, doi:10.1016/j.jss.2024.112253.
- [10] H. Dai, J. Xi, and H.-L. Dai, "Improve cross-project just-in-time defect prediction with dynamic transfer learning," *Journal of Systems and Software*, vol. 219, art. 112214, 2025, doi:10.1016/j.jss.2024.112214.

- [11] X. Ju, Y. Cao, X. Chen, L. Gong, V. Chakma, and X. Zhou, "JIT-CF: Integrating contrastive learning with feature fusion for enhanced just-in-time defect prediction," *Information and Software Technology*, vol. 182, art. 107706, 2025, doi:10.1016/j.infsof.2025.107706.
- [12] X. Liu, Y. Zhou, Y. Tang, J. Qian, and Y. Zhou, "Human-in-the-loop online just-in-time software defect prediction: What have we achieved and what do we still miss?" *Science of Computer Programming*, vol. 244, art. 103296, 2025, doi:10.1016/j.scico.2025.103296.
- [13] Q. Wu, X. Wang, D. Wei, B. Chen, and Q. Dang, "Just-in-time software defect prediction method for non-stationary and imbalanced data streams," *Software Quality Journal*, vol. 33, art. 14, 2025, doi:10.1007/s11219-025-09711-w.