# Deployment Risk Assessment Using Diff-Aware Features: A Case Study at Prime Video

Mayur Kurup*
kurupm@amazon.com
Amazon.com
Austin, Texas, USA

Hyunjae Suh*†
hyunjas@uci.edu
University of California
Irvine, California, USA

Swathi Vaidyanathan
vaiswath@amazon.com
Amazon.com
Seattle, Washington, USA

Pranesh Vyas
vyaspran@amazon.com
Amazon.com
Austin, Texas, USA

Srinidhi Madabhushi
srimadab@amazon.com
Amazon.com
Seattle, Washington, USA

Yegor Silyutin
silyutin@amazon.com
Amazon.com
Seattle, Washington, USA

## Abstract

At Amazon Prime Video, we face the critical operational challenge of managing code deployments during live events and rapid feature releases without causing service outages. Current change control approaches use blanket deployment freezes that block all changes regardless of risk, creating significant developer toil. While prior research has explored risky change predictors, these rely on developer-specific metadata or extensive historical data, raising privacy concerns and limiting applicability to new projects. We introduce a framework centered on diff-aware features—characteristics derived directly from code modifications. Our key contribution is the systematic identification of which quantitative metrics (code-level and change-level metrics) and qualitative indicators (coding style violations, change type classification) are necessary for risk prediction. We employ LLMs as multi-language feature extractors, demonstrating their effectiveness for code analysis beyond generation tasks and eliminating the need for language-specific tooling. We evaluated our framework on two datasets: Prime Video's production environment and the public ApacheJIT dataset. Our best-performing model achieves an average recall of **0.83** and F1 score of **0.81** across both datasets for detecting risky code changes. Notably, ablation analysis reveals that change-level volume metrics (e.g., lines added/deleted) are noisy predictors, while structural code complexity provides a substantially stronger risk signal. These results demonstrate that thoughtful feature curation enables effective change risk assessment across different programming languages and organizational contexts while avoiding privacy concerns.

## CCS Concepts

• **Software and its engineering → Software testing and debugging**; **Software maintenance tools**.

*Both authors contributed equally to this work
†Work done during Amazon internship


*ASE '26, Munich, Germany*


## Keywords

Change Control, Code Freeze, Change Risk, Proactive Safety, Just-in-Time Defect Prediction

**ACM Reference Format:**
Mayur Kurup, Hyunjae Suh, Swathi Vaidyanathan, Pranesh Vyas, Srinidhi Madabhushi, and Yegor Silyutin. 2026. Deployment Risk Assessment Using Diff-Aware Features: A Case Study at Prime Video. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26).* ACM, New York, NY, USA, 6 pages.

## 1 Introduction

Live event streaming services such as Prime Video operate under stringent reliability requirements, particularly during large-scale broadcasts like NFL Thursday Night Football (TNF) and Premier League matches, where millions of viewers depend on uninterrupted service. Even a single faulty code change can cause widespread disruptions that are difficult to mitigate during events. To reduce this risk, organizations commonly employ broad deployment freezes during high-traffic windows. While these freezes prevent failures, they delay feature rollouts, create pending change accumulations, and require special handling for urgent updates. As the number and complexity of live events continue to grow, the operational cost of such freezes increases correspondingly, making more precise assessment of individual code-change risk increasingly important.

The central challenge is determining which changes warrant caution without relying on coarse, event-long freezes. Although prior research on change- and defect-prediction provides useful insights [8, 14], many approaches depend on developer-specific metadata or substantial historical project information. These dependencies raise privacy considerations, hinder adoption across diverse teams, and limit applicability to new or rapidly evolving codebases. Other models demonstrate effectiveness only within a single dataset, providing limited evidence of generalizability across languages, repositories, or organizational environments.

These limitations motivate an approach using only code modification information. Diff-level characteristics avoid sensitive metadata, apply immediately to new projects including cold-start scenarios such as new code bases, and behave consistently across heterogeneous environments. Such a model would be particularly valuable in contexts like Prime Video's live event operations, where accurate

and lightweight risk assessment is needed to make informed deployment decisions during TNF and other high-visibility broadcasts.

In this work, we investigate the predictive utility of diff-aware features for assessing risky code changes. Our intended deployment model is as a *guardrail*: the system flags potentially risky changes for human review rather than autonomously blocking deployments, keeping engineers in the loop for final deployment decisions. This human-in-the-loop design motivates our preference for recall over precision: it is preferable to surface a change for review unnecessarily than to miss a genuinely risky one, leading us to select a threshold that weights recall more (Section 3.3). By constructing a comprehensive set of quantitative and qualitative attributes derived directly from code modifications, and evaluating them across both industrial and public datasets, we assess whether code-change information alone can support practical and reliable risk prediction. Our results indicate that a well-curated set of diff-level features captures meaningful risk signals, enabling more targeted and efficient deployment decision-making in environments where reliability and development velocity must be jointly balanced.

Our contributions are: (1) a diff-aware risk assessment framework applicable to cold-start scenarios without developer metadata, (2) LLM-based multi-language feature extraction that replaces brittle language-specific AST parsers, and (3) empirical evidence that LLMs are effective for structured feature extraction but lag behind feature-engineered models for end-to-end risk classification.

The remainder of this paper is organized as follows: Section 2 reviews related work, Section 3 describes our methodology including data collection, feature extraction, and classification, Section 4 presents the classification and ablation results, Section 5 discusses industrial deployment considerations and feature insights, Section 6 addresses threats to validity, and Section 7 concludes with future directions.

## 2 Related Work

Just-in-time (JIT) defect prediction aims to estimate whether a code change is likely to introduce a defect at the moment it is submitted. Kim et al. introduced change classification using features derived from change size, diffusion, and file entropy [10], establishing the per-commit granularity as a practical setting for early defect detection. Kamei et al. conducted large-scale empirical studies demonstrating that inspecting a small portion of commits can prevent a substantial fraction of defect-inducing changes [8], with change size and complexity metrics (diff-aware features) emerging as the strongest predictors while developer experience (diff-opaque) provided complementary signal. Their cross-project analysis [7] showed JIT models can transfer across projects under certain conditions, relevant to cold-start scenarios where historical data is unavailable. Deep learning approaches have been applied to learn semantic representations: Hoang et al. proposed DeepJIT, jointly encoding commit messages and diffs using CNNs [6], with ablation studies showing code-level features alone achieved 85–90% of full model performance, demonstrating that diff-aware features carry substantial predictive power independently. Pre-trained models such as CodeBERT [4] have been adapted to defect prediction, but these approaches use models as end-to-end classifiers requiring fine-tuning and large corpora. In contrast, we use LLMs strictly as feature extractors—leveraging their multi-language understanding to compute structured metrics, then feeding those into lightweight ML classifiers. This avoids the cost and opacity of end-to-end models while retaining language generalization. Other research investigates finer granularity: Pascarella et al. explored file-level JIT prediction [15], and Zeng et al. highlighted reproducibility issues in deep JIT approaches [19].

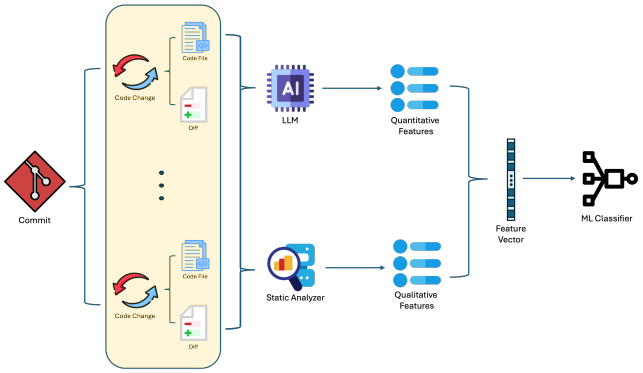


**Figure 1: Methodology Overview**

Despite extensive work, many models depend on developer-specific metadata, detailed historical traces, or computation-heavy deep learning pipelines. Such dependencies create challenges for industrial adoption, particularly where privacy constraints and rapid onboarding of new projects limit data availability. Our work complements prior research by focusing strictly on *diff-only* features, enabling a metadata-free and lightweight approach suitable for operational settings such as Prime Video's live-event streaming environment. Our findings validate that diff features alone can achieve strong performance (F1: 0.846), directly addressing cold-start scenarios common in industrial settings with frequent repository creation and strict privacy requirements.

## 3 Methodology

We describe our methodology, spanning data collection, feature extraction, and classification. Figure 1 presents an overview of the framework that extracts diff-aware features from commits using LLMs for quantitative metrics and static analyzers for qualitative indicators, then applies ML classifiers to predict change risk.

### 3.1 Data Collection

We collected ground truth data by linking Correction of Error (CoE)[1] reports, a mandatory post-incident review process in which engineers document root causes and link the specific commits responsible, to the corresponding commits that directly triggered the incident relating to the outage. Starting with 377 CoEs flagged as change-induced, we parsed each report to extract the Code Reviews (CRs) identified as root causes. Extracting structured information from CoE reports was challenging due to inconsistent formatting across teams. We used Claude Sonnet 3.7 to parse these reports and extract commit references. Domain experts then manually verified all extracted CoE-to-commit linkages from 2024 to ensure correct attribution, confirming that each linked commit was indeed the

root cause identified in the corresponding CoE report. We ranked packages by the number of associated CRs and selected the top 40 packages with the highest CR occurrences, yielding a total of 100 CRs. For each CR, we collected the corresponding commit linked to it. These commits form our ground truth dataset of risky changes.

To focus on widely-used languages in our dataset, we selected Java, Kotlin, and TypeScript for our experiments, which are the predominant languages in the Prime Video codebase. After analyzing our deployment data, we found that approximately 5% of commits to production result in CoE-flagged incidents. We prioritized labeling precision over dataset size, retaining only commits with high-confidence CoE linkage, and maintained the natural 1:19 risky-to-safe ratio, resulting in 149 risky and 2,831 non-risky commits (2,980 total) that reflect realistic production conditions. For external validation, we used the ApacheJIT dataset [9], using the most recent three years of data across all 14 Apache projects and sampling 1,115 bug-inducing and 21,185 clean commits (22,300 total) to maintain the 1:19 class ratio.

## 3.2 Feature Extraction

This study leverages diff-aware features extracted from software changes to evaluate change riskiness. We focus on diff-aware features for the following reasons:

- **Direct Causality to Change:** Diff-aware features have clear causal connections to potential issues, as changes in code structure, size, and complexity directly impact risk.
- **Immediate Applicability:** Diff-aware features apply to new projects or repositories without historical data, supporting first-time contributors and scenarios with limited project history.
- **Lower Maintenance:** Diff-aware features eliminate the need to maintain historical data or complex metadata, simplifying implementation and reducing computational overhead.

While diff-opaque features (e.g., developer experience, historical defect rates) could potentially enhance performance, they raise privacy concerns and limit generalizability; exploring their integration represents a promising direction for future work. We extracted 28 features categorized into quantitative and qualitative groups. Quantitative features include: (1) *code-level metrics* capturing file structure (lines of code, comments, blank lines, declarations), complexity (cyclomatic complexity, maximum nesting depth), and scope (class, method, and function counts with visibility modifiers); and (2) *change-level metrics* measuring modification extent (added/deleted/modified lines and chunks, churn, files changed). Qualitative features include change type classification (Adaptive, Corrective, Perfective, Preventive) derived from commit messages and coding style violations detected across 13 categories (e.g., naming conventions, code organization, documentation issues).

To extract quantitative features, we employed Claude Sonnet 3.7, which generalizes across multiple languages (Java, Kotlin, TypeScript) without language-specific reimplementation. We initially developed language-specific scripts using regular expressions and AST patterns, but maintaining separate extractors proved brittle; we therefore adopted LLM-based extraction, retaining the scripts as validation baselines. We validated LLM-extracted features against script-based ground truth (n=60 samples across Java/Kotlin/TypeScript) using two metrics: Pearson correlation [16], which measures linear agreement in magnitude and trend, and Intraclass Correlation Coefficient (ICC) [17], which assesses measurement reproducibility. The validation achieved Pearson correlation of 0.730 and ICC of 0.861 [12].

**Table 1: Comparative Analysis Results**

| Metric | Average |
|---|---|
| Pearson Correlation | 0.730 |
| Intraclass Correlation (ICC) | 0.861 |

Table 1 presents the comparison results between the feature values from the script and the LLM. The Pearson correlation coefficient and ICC indicate a strong linear relationship between the two sets of values. These results demonstrate that while LLM-extracted values may not be identical in magnitude to script-generated ones, they capture underlying trends and relative patterns with reasonable reliability, making LLMs a viable alternative for feature extraction in change risk assessment.

**Table 2: Style Categories**

| Category | Description |
|---|---|
| Annotations | Annotation usage and placement. |
| Block Checks | Rules for braces and block style. |
| Class Design | Class structure and design rules. |
| Coding | General coding practice checks. |
| Headers | File and license header requirements. |
| Imports | Rules for import order, unused imports, and duplicates. |
| Comments | API documentation format and completeness. |
| Metrics | Complexity and size measurements. |
| Miscellaneous | Other uncategorized checks. |
| Modifiers | Access and modifier order/usage. |
| Conventions | Naming rules for classes, methods, and variables. |
| Regexp | Regex-based custom checks. |
| Size Violations | Limits on line, method, or class length. |
| Whitespace | Spacing, indentation, and line break rules. |

For extracting the type of change, we adopted categories from previous works [11] (Adaptive, Corrective, Perfective, Preventive). We prompted Claude Sonnet 3.7 to classify the type of change into one of the four categories. To verify the correctness of the LLM's output for this specific feature, we performed stratified sampling across all four categories (n=50) with independent annotation by domain experts.

To extract style violations, we employed static analyzers rather than LLMs, as style violation identification requires consistency and objectivity. We used language-specific analyzers: Checkstyle for Java [2], Detekt for Kotlin [3], and ESLint for TypeScript [18]. Since each analyzer uses different rulesets, we manually mapped all rules to common categories (Table 2) based on official documentation, ensuring uniformity across languages. For model input, we counted violation occurrences per category and formatted them as numerical features.

### 3.3 Classification

We applied machine learning models to classify change riskiness. For commits with multiple file modifications, we created unified commit-level representations by averaging quantitative features across files and diffs, and averaging violation counts across files for qualitative features.

We applied XGBoost and Random Forest classifiers, evaluated using 10-fold cross-validation within each dataset separately with hyperparameter tuning via grid search. Because our framework is designed as a guardrail that flags changes for human review rather than blocking them autonomously, we favor recall over precision: missing a risky change is costlier than surfacing a safe one for review. To select the classification threshold, we performed a grid search over candidate thresholds $\{0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5\}$ *within* each cross-validation fold to prevent information leakage, optimizing for F1-score. The threshold of 0.2 consistently maximized F1 across folds: given the 1:19 class imbalance, the default 0.5 threshold yields low recall, and lowering the threshold to 0.2 recovers sufficient true positives to improve F1 despite the precision trade-off. Lower thresholds (e.g., 0.1) yielded marginal recall gains at disproportionate precision cost, while higher thresholds (e.g., 0.3+) sacrificed recall without meaningful precision improvement. We evaluated models using precision, recall, and F1-score. As a baseline, we also performed Logistic regression and zero-shot and few-shot prompting with Claude Sonnet 3.7 and Sonnet 4.5, providing the model with the commit diff, commit description, extracted diff-aware features, and business context for risk classification. We additionally evaluated Claude Sonnet 4.5 under both settings to assess whether newer LLMs narrow the gap with feature-engineered models. The few-shot configuration included three risky and three safe labeled examples, stratified across packages to avoid bias, selected randomly with a fixed seed and excluded from evaluation.

We conducted an ablation study to identify the most influential feature groups using our best-performing model. We tested six configurations: three that excluded one feature group while retaining the other two (Quantitative Code-Level, Quantitative Change-Level, or Qualitative), and three that used only a single feature group to isolate its individual impact.

## 4 Results

### 4.1 Classification Results

We trained and evaluated our models using 10-fold cross-validation on both Prime Video and ApacheJIT datasets, consistent with prior JIT studies [6, 8], with results presented in Table 3. XGBoost achieved F1-scores of 0.846 (±0.070) on Prime Video and 0.771 (±0.048) on ApacheJIT. On Prime Video, it attained recall of 0.788 with precision of 0.926; on ApacheJIT, recall of 0.875 with precision of 0.689, reflecting different dataset characteristics. The higher variance on Prime Video is expected given its smaller dataset size (2,980 vs. 22,300 commits). The Logistic Regression baseline achieved F1-scores of 0.145 on Prime Video and 0.158 on ApacheJIT.

The Random Forest model achieved comparable F1-scores (Prime Video: 0.815, ApacheJIT: 0.746). On Prime Video, it outperformed XGBoost in recall (0.830) but with lower precision (0.811). On ApacheJIT, it achieved exceptional recall of 0.978, though with lower precision (0.603).

**Table 3: Model Performance Comparison (10-fold CV)**

| Model | Prime Video | | | ApacheJIT† | | |
|---|---|---|---|---|---|---|
| | Prec. | Rec. | F1 | Prec. | Rec. | F1 |
| LR (Baseline) | 0.116 | 0.199 | 0.145 | 0.124 | 0.218 | 0.158 |
| XGBoost | **0.926** | 0.788 | **0.846** | **0.689** | 0.875 | **0.771** |
| RandomForest | 0.811 | **0.830** | 0.815 | 0.603 | **0.978** | 0.746 |
| Sonnet 3.7 (Zero-Shot) | 0.087 | 0.641 | 0.153 | 0.098 | 0.810 | 0.175 |
| Sonnet 3.7 (Few-Shot) | 0.137 | 0.615 | 0.216 | 0.115 | 0.885 | 0.204 |
| Sonnet 4.5 (Zero-Shot) | 0.135 | 0.792 | 0.231 | 0.134 | 0.800 | 0.230 |
| Sonnet 4.5 (Few-Shot) | 0.157 | 0.779 | 0.261 | 0.127 | 0.872 | 0.222 |

† LLM-based classifiers were evaluated on a stratified subsample of 2,980 ApacheJIT commits (149 buggy, 2,831 clean) matching the Prime Video dataset size to enable cost-effective comparison. ML models were evaluated on the full dataset.

LLM-based classification with Claude Sonnet 3.7 achieved F1 of 0.153 (zero-shot) and 0.216 (few-shot) on Prime Video. Upgrading to Claude Sonnet 4.5 improved few-shot performance to 0.157 precision and 0.779 recall (F1: 0.261) on Prime Video and 0.127 precision and 0.872 recall (F1: 0.222) on ApacheJIT. While still below the feature-engineered XGBoost model (F1: 0.846), LLM-based classification shows promise with future steps such as fine-tuning on Prime Video's service dependency graph, incorporating diff-opaque features, and hybrid LLM-ML architectures.

Direct comparison with prior JIT models is limited because most report AUC—which averages over all thresholds and error types—rather than threshold-specific metrics suited to our high-recall guardrail setting, and incorporate developer metadata unavailable in our diff-only approach.

### 4.2 Ablation Study

To understand the contribution of each feature group, we performed an ablation study using XGBoost. We tested six variants: three that excluded one feature group while retaining the others, and three that used only a single feature group.

**Table 4: Ablation Study Results**

| Category | Prec. | Rec. | F1 |
|---|---|---|---|
| Baseline | 0.926 | 0.788 | 0.846 |
| Exclude Quant. Code-Level | 0.873 | 0.736 | 0.781 |
| Exclude Quant. Change-Level | 0.888 | **0.822** | **0.848** |
| Exclude Qualitative | **0.931** | 0.762 | 0.832 |
| Quant. Code-Level Only | 0.853 | 0.778 | 0.809 |
| Quant. Change-Level Only | 0.541 | 0.608 | 0.565 |
| Qualitative Only | 0.759 | 0.737 | 0.743 |

As presented in Table 4, the full model with all feature groups achieves the best performance. Excluding Quantitative Code-Level features caused the largest performance drop (F1: 0.781 vs. baseline 0.846), demonstrating their critical importance. Excluding Quantitative Change-Level features slightly improved performance (F1: 0.848), suggesting these features may introduce noise; we retained them because a marginal gain of 0.002 is unlikely to generalize. Excluding Qualitative features resulted in the highest precision (0.931) but lower recall (0.762) and F1 (0.832), indicating these features help balance the model's predictive capabilities.

Among single-group variants, Quantitative Code-Level Only achieved the best F1 (0.809), confirming code-level metrics as the most influential predictors. Qualitative Only maintained reasonable performance (F1: 0.743), while Quantitative Change-Level Only exhibited severe degradation (F1: 0.565), suggesting these features provide limited value in isolation. These results indicate that structural code complexity, not raw change volume, is the primary driver of risk prediction—a practical insight for teams prioritizing which features to invest in when building change risk systems.

## 5 Discussion

*Industrial Deployment and Operational Impact.* Our approach enables selective deployment decisions at the commit level during live events, potentially reducing freeze scope and accelerating feature velocity. Deployment freezes currently block 100% of changes regardless of risk, though only approximately 5% of commits historically cause incidents. On held-out 2024 production data, the model achieved 0.926 precision and 0.788 recall, suggesting it could clear the vast majority of safe changes for deployment while flagging most risky ones for human review—substantially reducing freeze scope without compromising safety. The threshold is tunable: lowering it increases recall further at modest precision cost, acceptable for a system that flags for review rather than auto-blocks. Adoption barriers include CI/CD integration, threshold calibration against business impact, and developer trust through transparent explanations. Validating these estimates through shadow-mode deployment during live events is planned as immediate next work.

*Trade-offs and Design Constraints.* While combining diff-aware features with historical metadata could enhance performance, Prime Video's operational constraints make a diff-only approach optimal: microservice churn makes historical data stale, cross-functional onboarding creates cold-start scenarios, and diff-aware features preserve privacy. Our ablation study (F1: 0.846) suggests diminishing returns from metadata enrichment may not justify maintenance costs.

A key practical benefit of LLM-based feature extraction is eliminating language-specific static analysis tooling. Traditional approaches require separate AST parsers and extraction scripts for each language, each maintained independently as languages evolve. A single LLM extractor generalizes across languages with one prompt template, significantly lowering the cost of onboarding new languages or repositories.

*Feature Contribution Insights.* A counterintuitive finding emerged from our ablation study: quantitative change-level features (lines added/deleted, chunks modified) degraded model performance, suggesting raw change volume is a noisy proxy for risk. Code-level structural complexity and qualitative indicators provide more reliable signals, confirmed by the ablation results (complexity-only F1: 0.809 vs. size-only F1: 0.565).

These LLM baselines represent the state of the art for end-to-end agentic risk classification without feature engineering. Performance improved from F1 0.153 to 0.231 (zero-shot) and 0.216 to 0.261 (few-shot) across model generations, demonstrating that newer models reduce the gap but do not close it—structured feature curation remains essential. This gap underscores that decomposing diffs into interpretable quantitative and qualitative dimensions captures risk patterns that end-to-end LLM classification misses, while also producing explainable predictions suitable for developer-facing guardrails.

ApacheJIT's lower F1-score (0.771 vs 0.846) likely stems from dataset characteristics: Prime Video's CoE-based labeling versus ApacheJIT's diverse post-release defect labeling. Our findings extend prior observations that code-level features carry substantial predictive power [6, 8] to a diff-only standalone model for cold-start scenarios [5, 13].

## 6 Threats to Validity

While our models demonstrate strong performance across both industrial (Prime Video) and open-source (ApacheJIT) datasets, results may be influenced by dataset-specific characteristics.

*Construct Validity.* Prime Video uses CoE reports while ApacheJIT uses post-release defects as ground truth, representing different operationalizations of "risky change." The Prime Video dataset is relatively small (149 risky commits) because not all faulty changes result in CoE reports, and high-confidence ground truth filtering further reduces the set. Domain experts manually verified all CoE-to-commit linkages from 2024, confirming correct attribution for all reviewed cases.

*Internal Validity.* Our feature set may not capture all aspects of a software change. LLM extraction noise (ICC: 0.861) is absorbed during model fitting since it applies identically at train and inference time; cross-validated F1 inherently reflects this. We evaluated LLM-based classification under zero-shot and few-shot settings with two model generations; few-shot yielded modest improvement, with larger gains on newer models (Sonnet 4.5), though fine-tuning could further narrow the gap. We use 10-fold cross-validation rather than temporal splits, consistent with prior JIT work [8, 15]; temporal evaluation is future work.

*External Validity.* The language distribution (Java, Kotlin, TypeScript) may limit feature robustness for underrepresented languages. We mitigated this by evaluating across two distinct datasets with different languages, organizational contexts, and labeling practices. Our features remain applicable to AI-generated code, as structural complexity and style violations characterize the change itself regardless of authorship.

## 7 Conclusion and Future Work

We introduced a diff-aware framework for change risk assessment that derives features directly from code modifications, avoiding dependencies on developer metadata or historical data. On Prime Video and ApacheJIT, XGBoost achieved F1-scores of 0.846 and 0.771, with code-level features proving most influential. Future work includes: (1) shadow-mode deployment during live events to validate predictions against real outcomes, followed by A/B testing to measure operational impact, (2) hybrid models with organizational metadata, (3) semantic change analysis, and (4) fine-tuned LLMs, as zero-shot and few-shot prompting showed limited effectiveness, though Sonnet 4.5 improved few-shot F1 by 21% over Sonnet 3.7, suggesting LLM-based classification is a viable direction with further investment.

## 8 Data Availability Statement

The primary dataset in this study is derived from Amazon's proprietary production systems and cannot be released due to the confidentiality of internal source code, deployment records, and operational data. To support external validation, we additionally evaluate on the publicly available ApacheJIT dataset [9]. All diff-aware features are defined in full in Section 3, enabling reproduction of our feature-extraction pipeline on other codebases.

## References


[1] Inc. Amazon Web Services. 2022. *Why you should develop a correction of error (COE).* https://aws.amazon.com/blogs/mt/why-you-should-develop-a-correction-of-error-coe Accessed: 2026-07-01.

[2] Checkstyle. 2025. Checkstyle – A Static Analysis Tool for Java. https://checkstyle.org/. Accessed: 2025-09-05.

[3] Detekt. 2025. Detekt – A Static Code Analysis Tool for Kotlin. https://detekt.dev/. Accessed: 2025-09-05.

[4] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020). https://doi.org/10.18653/v1/2020.findings-emnlp.139

[5] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. 2014. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*. ACM, 172–181. https://doi.org/10.1145/2597073.2597075

[6] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 34–45. https://doi.org/10.1109/MSR.2019.00016

[7] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. 2016. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering* 21, 5 (2016), 2072–2106. https://doi.org/10.1007/s10664-015-9400-x

[8] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 757–773. https://doi.org/10.1109/TSE.2012.70

[9] Hossein Keshavarz and Meiyappan Nagappan. 2022. *ApacheJIT: A Large Dataset for Just-In-Time Defect Prediction.* https://doi.org/10.5281/zenodo.5907002

[10] Sunghun Kim, E James Whitehead, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on software engineering* 34, 2 (2008), 181–196. https://doi.org/10.1109/TSE.2007.70773

[11] Stanislav Levin and Amiram Yehudai. 2017. Boosting Automatic Commit Classification Into Maintenance Activities By Utilizing Source Code Changes. In *13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. 97–106. https://doi.org/10.1145/3127005.3127016

[12] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. Assisting Static Analysis with Large Language Models: A ChatGPT Experiment. In *Proceedings of the 31st ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23)*. ACM, New York, NY, USA. https://doi.org/10.1145/3611643.3613078

[13] Xutong Liu, Zhiyu Yan, Zhenyue Chen, Peng Xu, and Yibiao Yang. 2025. 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* (2025). https://doi.org/10.1016/j.scico.2025.103296

[14] Audris Mockus and David M. Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* 5, 2 (2000), 169–180. https://doi.org/10.1002/bltj.2229

[15] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2019. Fine-grained just-in-time defect prediction. *Journal of Systems and Software* 150 (2019), 22–36. https://doi.org/10.1016/j.jss.2018.12.001

[16] Karl Pearson. 1895. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London* 58 (1895), 240–242. https://doi.org/10.1098/rspl.1895.0041

[17] Patrick E Shrout and Joseph L Fleiss. 1979. Intraclass correlations: Uses in assessing rater reliability. *Psychological Bulletin* 86, 2 (1979), 420–428. https://doi.org/10.1037/0033-2909.86.2.420

[18] typescript-eslint. 2025. typescript-eslint – ESLint and Prettier Support for TypeScript. https://typescript-eslint.io/. Accessed: 2025-09-05.

[19] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. 2021. Deep just-in-time defect prediction: how far are we?. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*. 427–438. https://doi.org/10.1145/3460319.3464819