International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V5I4P122
Eureka Vision Publication | Volume 5, Issue 4, 192-202, 2024

Original Article

Machine Learning-Based Prediction of Technical Debt in Continuous Software Development Using Repository Mining and Developer Activity Analytics

Dr. Deepshikha Tiwari[1], Dr. Shivani Sharma[2], Dr. Kapil Tomar[3], Dr. Saurabh Arora[4]
[1,2]Assistant Professor, CSE, Thapar Institute of Engineering & Technology, Patiala, Punjab, India.
[3,4]Assistant Professor, AI & ML, Thapar Institute of Engineering & Technology, Patiala, Punjab, India.

Abstract - Technical debt is no longer an occasional by-product of rushed implementation; it has become a persistent socio-technical risk in continuous software development environments where code, build scripts, configuration artifacts, tests, issue tickets, pull requests, and deployment workflows evolve at high velocity. Conventional technical debt management methods often depend on manual code reviews, static analysis alerts, or retrospective refactoring decisions, which are insufficient for detecting emerging debt before it becomes costly to repay. This paper proposes a machine learning-based framework for predicting technical debt in continuous software development by integrating repository mining, developer activity analytics, change-process metrics, self-admitted technical debt signals, and software lifecycle governance indicators. The proposed framework models technical debt as an evolving risk state rather than a static code smell, combining structural code metrics, commit-level churn, ownership dispersion, pull-request review behavior, issue-tracker semantics, CI/CD instability, and developer workload signals. The study introduces a multi-source feature architecture, a temporal labeling strategy, an explainable ensemble learning pipeline, and a debt-prioritization layer designed for continuous integration environments. Unlike review-oriented approaches that summarize technical debt literature, this manuscript develops a research-oriented predictive design that can be operationalized in modern software engineering pipelines. The expected contribution is a decision-support mechanism that helps engineering teams identify debt-prone modules, anticipate repayment urgency, improve sprint planning, and align refactoring decisions with delivery risk, developer capacity, and long-term maintainability.

Keywords - Technical Debt Prediction, Repository Mining, Developer Analytics, Machine Learning, Continuous Software Development, Self-Admitted Technical Debt, Software Maintenance, CI/CD Risk, Explainable AI.

1. Introduction

Continuous software development has changed the economics of technical debt. In traditional release cycles, debt often accumulated through visible architectural shortcuts, delayed documentation, weak test coverage, or duplicated implementation decisions. In modern continuous delivery environments, debt can form more subtly through rapid pull-request merging, fragmented ownership, unstable build scripts, emergency patches, postponed refactoring, and shifting dependencies. Because these debt signals are distributed across repositories, issue trackers, build logs, review platforms, and developer activity traces, a purely code-centric technical debt assessment is incomplete. This paper argues that technical debt prediction must be modeled as a socio-technical forecasting problem, where repository artifacts and developer behavior jointly reveal debt accumulation before it becomes visible as failure, delay, or maintainability loss.

The original technical debt metaphor emphasized the long-term cost of expedient implementation choices, especially when developers move forward with code that works temporarily but later requires expensive correction [17]. In contemporary software systems, this metaphor remains relevant but must be extended to include continuous integration pipelines, distributed teams, automated testing practices, and machine learning-assisted engineering workflows. A debt-prone component may not simply be a complex class or a duplicated method. It may be a module with repeated emergency commits, low review depth, frequent ownership changes, unresolved TODO comments, unstable tests, and issue tickets that repeatedly mention workaround behavior.

Technical debt management has been studied through systematic mapping, self-admitted technical debt detection, architecture analysis, static quality assessment, and repository mining. However, many existing approaches remain either retrospective or limited to isolated artifact types. A repository-mining approach that only analyzes code churn may identify volatile files but miss developer overload and review bottlenecks. A self-admitted technical debt classifier that only scans comments may detect explicit debt but ignore silent debt emerging through build instability or dependency drift. A static analyzer may flag complexity but fail to distinguish harmful debt from acceptable complexity in actively maintained modules. Therefore, a predictive framework must integrate multiple signals and preserve temporal context.

The motivation for the proposed work is grounded in the observation that software engineering teams increasingly use machine learning and decision intelligence to improve lifecycle governance, defect prediction, automated testing, and project management [5]. These governance-oriented models suggest that engineering decisions can be strengthened when predictive analytics are embedded into agile workflows rather than applied after quality degradation has already occurred. In the same direction, intelligent CI/CD risk detection research shows that deployment pipelines can benefit from machine learning models that evaluate operational and release risk before production exposure [1]. Technical debt prediction should be treated similarly: not as an isolated maintenance concern but as a continuous risk signal that informs delivery governance.

The specific problem addressed in this paper is the lack of an integrated, explainable, continuous, and developer-aware model for predicting technical debt at the component level. The proposed approach introduces a framework named TD-DevPredict, which mines software repositories and developer activity streams to estimate debt likelihood, classify debt type, and prioritize repayment urgency. The framework combines code metrics, historical change behavior, self-admitted technical debt evidence, issue-tracker semantics, pull-request review features, CI/CD failure patterns, and developer activity indicators. The model is designed to support weekly or sprint-level prediction, enabling engineering teams to identify high-risk components before they become maintenance bottlenecks.

This paper makes four research contributions. First, it formalizes technical debt prediction as a temporal, multi-source classification and prioritization task rather than a one-time static analysis problem. Second, it proposes a feature architecture that integrates repository mining with developer activity analytics, including ownership concentration, review latency, workload distribution, and commit burstiness. Third, it presents an explainable ensemble learning design that supports debt probability estimation and debt-type interpretation. Fourth, it introduces a continuous integration deployment model that enables debt prediction to operate within agile development, sprint planning, release readiness, and refactoring prioritization.

2. Background and Research Gap

Technical debt research has developed across several complementary directions. Systematic mapping studies have organized the field around debt identification, measurement, monitoring, prioritization, repayment, and prevention [4]. These studies show that technical debt is a broad construct involving design debt, code debt, test debt, documentation debt, architecture debt, build debt, and requirement debt. However, mapping studies also reveal a persistent challenge: debt is difficult to quantify because it is partly technical, partly economic, and partly organizational. A module may contain debt not only because it has high complexity, but because its maintainers lack sufficient knowledge, its tests are brittle, or its business logic changes too frequently.

Self-admitted technical debt research introduced a practical path for debt detection by mining developer comments that explicitly acknowledge incomplete, temporary, or suboptimal implementation decisions [2]. This line of work is valuable because it captures developer intent. Developers often leave comments such as TODO, FIXME, workaround, temporary fix, or later refactor when they know a solution is imperfect. However, self-admitted debt is only a partial signal. Many harmful debt items are never documented in comments, and some comments remain after the debt has already been repaid. Therefore, SATD should be treated as one predictive feature group rather than the complete definition of debt.

Subsequent work categorized self-admitted technical debt into several types, including design debt, defect debt, documentation debt, requirement debt, and test debt [10]. This classification is important for prediction because different debt types have different causes and repayment strategies. Design debt may be associated with architectural shortcuts and high coupling, while test debt may be associated with missing coverage, flaky tests, or disabled assertions. Documentation debt may appear in issue trackers and pull requests rather than source code. A predictive model that only outputs a binary debt label provides limited value; a model that estimates debt type and repayment urgency is more useful for engineering decision-making.

Repository mining provides another foundation for technical debt prediction. Change metrics have been shown to be powerful indicators of future software quality because frequently modified files are more likely to accumulate defects, instability, and maintenance complexity [8]. Churn-based predictors are especially relevant in continuous development because technical debt often accumulates through repeated small modifications rather than one large architectural mistake. A file that receives many small patches from many developers may indicate unstable requirements, unclear ownership, or insufficient abstraction boundaries.

Just-in-time quality assurance research demonstrated that change-level prediction can help teams focus inspection effort on risky commits rather than reviewing all changes equally [6]. This insight is directly applicable to technical debt prediction. Instead of asking whether an entire repository has debt, the model should estimate whether a specific change, file, service, or component is increasing debt risk. Such change-aware prediction can reduce refactoring cost because debt is easier to address near the point of introduction than after it spreads across modules.

Developer activity analytics further expands the predictive space. Ownership, experience, and contribution distribution influence software quality because modules modified by many low-expertise or peripheral contributors may have higher defect and maintenance risk [14]. Technical debt is similarly affected by human and organizational factors. A technically simple module can become debt-prone if knowledge is fragmented, review ownership is unclear, or urgent fixes are repeatedly made by developers with limited context. Therefore, developer behavior should not be treated as peripheral metadata; it is a core signal for debt prediction.

The software engineering literature also shows that code ownership measures are associated with failures in large software systems [20]. This supports the argument that technical debt prediction should include ownership dispersion, top-contributor dominance, recent contributor churn, and reviewer familiarity. Debt is not only embedded in code; it is embedded in the mismatch between code complexity and team knowledge. When a complex module lacks stable ownership, the risk of delayed repayment and recurring workaround implementation increases.

Machine learning has already been used for defect prediction, software lifecycle governance, quality assurance, and technical debt identification. Comparative studies of machine learning models for software defect prediction indicate that model selection, feature representation, and dataset characteristics strongly influence predictive performance [23]. Technical debt prediction faces similar challenges because debt labels are noisy, imbalanced, and context-sensitive. A practical model must therefore combine robust feature engineering, temporal validation, explainability, and careful label construction.

Recent research on machine learning for technical debt identification evaluates statistical and machine learning classifiers over software metrics and repository activity features [25]. This direction is highly relevant but requires extension toward continuous development settings. Technical debt is not a static label assigned once to a class. It evolves as teams introduce features, modify tests, change ownership, and adjust deployment pipelines. Therefore, a continuous model must use rolling time windows, temporal leakage prevention, and sprint-aware prediction horizons.

The research gap is clear. Existing approaches provide valuable foundations for technical debt mapping, SATD detection, change-risk prediction, ownership analytics, and ML-based classification. However, there remains a need for a unified predictive framework that integrates code, process, developer, textual, and pipeline signals into an explainable model suitable for continuous software development. TD-DevPredict addresses this gap by modeling debt likelihood, debt type, and repayment priority through repository mining and developer activity analytics.

3. Problem Statement and Research Questions

The central problem is that engineering teams lack a reliable predictive mechanism for identifying technical debt before it becomes expensive to repay. Current technical debt management is often reactive. Teams discover debt during production incidents, delayed feature delivery, onboarding difficulty, failed regression testing, or developer complaints. By the time debt becomes visible, repayment may require cross-module refactoring, architectural redesign, test reconstruction, and stakeholder negotiation. A predictive system should instead identify emerging debt risk while the affected code is still actively changing.

This paper defines technical debt prediction as the task of estimating whether a software component, within a future development window, is likely to require costly corrective work due to accumulated implementation shortcuts, unstable evolution, inadequate tests, weak documentation, fragmented ownership, or pipeline instability. The prediction unit may be a file, class, package, microservice, repository subdirectory, or deployable component. The prediction horizon may be one sprint, one release cycle, or a configurable number of commits.

The proposed research is guided by the following research questions:

- RQ1: Which repository-mining features provide the strongest signals for predicting technical debt in continuous software development?
- RQ2: How do developer activity indicators such as ownership dispersion, review latency, contributor experience, and workload burstiness improve technical debt prediction?
- RQ3: Can a multi-source machine learning framework predict technical debt more effectively than isolated code-metric or SATD-only baselines?
- RQ4: How can explainability techniques convert technical debt predictions into actionable refactoring and sprint-planning recommendations?
- RQ5: How can technical debt prediction be integrated into CI/CD workflows without blocking delivery or creating excessive alert fatigue?

The working hypothesis is that technical debt can be predicted more effectively when repository-mining features and developer activity features are jointly modeled. Code churn, complexity, and SATD comments may indicate local debt, but developer analytics and pipeline behavior provide contextual signals about whether that debt is likely to persist, compound, or

affect delivery. This hypothesis is consistent with broader agile lifecycle governance models that combine predictive quality assurance and decision intelligence [21].

4. Proposed TD-DevPredict Framework

TD-DevPredict is a multi-layer framework for machine learning-based technical debt prediction in continuous development environments. The framework consists of seven major layers: data ingestion, artifact normalization, feature extraction, temporal labeling, model training, explainability, and workflow integration. Each layer is designed to support incremental execution so that debt predictions can be updated after commits, pull requests, builds, or sprint planning events.

The data ingestion layer collects repository artifacts from Git-based version control systems, issue trackers, pull-request platforms, static analysis tools, test reports, CI/CD logs, and developer activity records. This layer does not assume that a single artifact type contains the full debt signal. Instead, it treats technical debt as a multi-source phenomenon. For example, a code comment may identify a workaround, an issue ticket may describe delayed refactoring, a pull request may show repeated reviewer concern, and a failed build may indicate test or configuration debt.

The artifact normalization layer aligns data at the component and time-window level. Since technical debt evolves over time, the model must avoid mixing future information into past predictions. For each prediction window, the framework constructs a snapshot of component-level features based only on information available before the prediction date. This is essential because repository-mining studies can easily overestimate accuracy if they allow future bug fixes, later comments, or post-release labels to influence earlier predictions.

The feature extraction layer is the core technical contribution. It constructs five feature groups: structural code features, change-process features, SATD and textual features, developer activity features, and CI/CD stability features. Structural features include complexity, coupling, size, duplication, dependency depth, and static rule violations. Change-process features include churn, number of commits, number of modified files, patch size, refactoring frequency, revert frequency, and temporal burstiness. Textual features include TODO/FIXME density, workaround keywords, issue-topic embeddings, and pull-request discussion semantics. Developer activity features include ownership concentration, recent contributor count, reviewer familiarity, review latency, and contributor workload. CI/CD features include failed build ratio, flaky test evidence, test execution duration, deployment rollback count, and pipeline configuration changes.

The temporal labeling layer defines technical debt outcomes. Since technical debt labels are rarely available in clean form, the framework supports three labeling strategies. The first is explicit labeling through SATD comments, issue tags, and maintenance tickets. The second is repayment labeling, where future refactoring commits, debt-removal comments, or test reconstruction work indicate that prior debt existed. The third is expert-confirmed labeling, where maintainers validate debt candidates generated by static analysis and repository heuristics. This hybrid labeling design reduces dependence on any single imperfect source.

The model training layer uses an ensemble architecture. Tree-based models such as random forests, gradient boosting, and extreme gradient boosting are suitable for heterogeneous structured features. Text embeddings from issue tickets, comments, and pull-request discussions can be incorporated through transformer-derived vectors or lightweight TF-IDF representations depending on resource constraints. The final model estimates debt probability, debt category, and repayment urgency. Ensemble modeling is appropriate because technical debt signals are nonlinear and distributed across artifacts.

The explainability layer uses feature attribution, local explanations, and rule-based summarization. Engineering teams are unlikely to trust a technical debt model that only outputs a probability score. They need to know why a component is considered risky. For example, a prediction may be explained by high churn, multiple emergency commits, low test coverage, rising review latency, and repeated TODO comments. Explainability also supports governance because teams can decide whether to repay, monitor, defer, or accept the debt.

The workflow integration layer connects predictions to sprint planning, pull-request review, release readiness, and refactoring backlog management. The model should not block every risky change. Instead, it should assign risk levels and recommend actions. Low-risk debt can be monitored. Medium-risk debt can be assigned to backlog refinement. High-risk debt can trigger architecture review, additional testing, or refactoring before release. This approach aligns with cloud-native and microservices governance practices where automated analysis supports, but does not replace, engineering judgment [3].

5. Feature Engineering for Repository Mining and Developer Analytics

Technical debt prediction depends heavily on feature quality. TD-DevPredict organizes features into interpretable groups so that model outputs can be explained to developers, technical leads, and engineering managers.

5.1. Structural Code Features

Structural features describe the current state of the codebase. These include lines of code, cyclomatic complexity, nesting depth, coupling between objects, cohesion, fan-in, fan-out, inheritance depth, method length, class size, duplicate code ratio, dependency count, and static rule violations. These metrics are useful because debt often appears as code that is hard to understand, test, modify, or isolate.

However, structural features alone can be misleading. A complex component may be stable, well-tested, and maintained by experienced developers. Conversely, a small component may be risky if it changes frequently, lacks tests, and is modified by many contributors. Therefore, TD-DevPredict treats structural features as necessary but insufficient indicators.

5.2. Change-Process Features

Change-process features capture how code evolves. These include commit frequency, churn volume, number of deleted and added lines, file co-change frequency, average patch size, hotfix count, revert frequency, and time since last major refactoring. Change metrics are central to debt prediction because technical debt often accumulates through repeated short-term modifications. Relative churn measures are especially useful because absolute change size alone may not reflect risk [12].

The framework also includes temporal burstiness, defined as the concentration of changes within short periods. A module modified intensely before deadlines may accumulate rushed decisions, insufficient tests, or partial refactoring. Similarly, repeated small commits after failed builds may signal unstable implementation. These features allow the model to detect debt formation patterns rather than only static code conditions.

5.3. Self-Admitted Technical Debt and Textual Features

Textual features are mined from source comments, issue trackers, commit messages, and pull-request discussions. Comments containing terms such as TODO, FIXME, workaround, temporary, hack, later, refactor, brittle, or cleanup may indicate explicit developer awareness of debt. Text mining approaches have shown that SATD can be identified from large sets of comments with higher accuracy than simple keyword matching when classifier-based methods are used [18].

Issue trackers and pull requests provide additional signals. Developers often discuss technical compromises in issue comments rather than source code. A pull request may mention that a solution is temporary because a release deadline is near. An issue may repeatedly reopen due to incomplete fixes. TD-DevPredict transforms these textual signals into features using keyword patterns, topic modeling, sentence embeddings, and debt-type classifiers.

5.4. Developer Activity Features

Developer activity features are designed to capture socio-technical risk. They include number of unique contributors per component, ownership concentration, top-owner share, recent contributor turnover, average contributor experience, reviewer familiarity, review comment density, review latency, and after-hours commit ratio. These features reflect how development behavior affects maintainability.

A component with stable ownership and experienced reviewers may tolerate complexity better than a component with fragmented responsibility. Conversely, a module touched by many occasional contributors may accumulate inconsistent patterns and hidden assumptions. This is especially important in agile teams where rapid onboarding, distributed development, and parallel feature work can weaken architectural coherence.

5.5. CI/CD and Operational Features

Technical debt often manifests in build and deployment instability. CI/CD features include build failure frequency, test failure frequency, flaky test ratio, pipeline duration, rollback count, deployment delay, environment-specific failures, and changes to build configuration. Continuous delivery studies and intelligent deployment-risk models suggest that software quality and operational risk can be predicted from pipeline signals [1].

Build debt and test debt are often underrepresented in code-only approaches. A codebase may look clean while its pipeline is fragile, tests are slow, or deployment scripts require manual intervention. TD-DevPredict therefore treats CI/CD instability as a first-class debt signal. This is particularly important for microservices and cloud-native systems where configuration, deployment, and observability debt can cause significant operational risk.

6. Machine Learning Methodology

The proposed methodology follows a temporal supervised learning design. Each component is represented by a feature vector computed at time t, and the model predicts whether technical debt will be observed, repaid, escalated, or confirmed within a future time window $t + h$. The horizon h may be configured as one sprint, four weeks, one release cycle, or a fixed number of commits.

The baseline models include logistic regression, decision trees, random forests, support vector machines, gradient boosting, and naïve Bayes for textual features. More advanced models include XGBoost, LightGBM, graph-based dependency models, and hybrid architectures that combine structured metrics with textual embeddings. Ensemble models are suitable because technical debt is multi-causal and nonlinear. Prior defect prediction research demonstrates that different machine learning models can behave differently across datasets, reinforcing the need for comparative evaluation rather than relying on a single algorithm [28].

The proposed model pipeline includes data cleaning, feature normalization, categorical encoding, imbalance handling, temporal train-test splitting, hyperparameter tuning, and explainability extraction. Imbalance handling is particularly important because confirmed technical debt events may be less frequent than non-debt observations. The framework supports cost-sensitive learning, class weighting, and threshold tuning to balance precision and recall. For technical debt management, false negatives may be costly because missed debt can compound, while false positives can create alert fatigue.

The labeling process is one of the most difficult aspects of the methodology. TD-DevPredict uses a hybrid labeling strategy. A positive debt label can arise from explicit SATD comments, issue labels, refactoring commits, debt-removal comments, static analysis violations confirmed by maintainers, or repeated maintenance tickets associated with the same component. A negative label is assigned only when a component shows stable maintenance behavior, no debt evidence, no debt-related issues, and no repayment activity within the horizon. Ambiguous cases are excluded or treated as weak labels.

The framework also supports debt-type classification. The debt categories include design debt, code debt, test debt, documentation debt, build debt, requirement debt, architecture debt, and operational debt. These categories reflect the reality that not all debt should be repaid in the same way. Test debt requires test reconstruction, build debt requires pipeline stabilization, and architecture debt may require design review. Research on issue-tracker SATD has identified debt types beyond code comments, reinforcing the need for multi-source classification [22].

Explainability is implemented through global and local analysis. Global feature importance identifies the strongest overall predictors, such as churn, ownership dispersion, review latency, or build failure rate. Local explanations describe why a specific component was flagged. For example, the model may generate an explanation such as: “Component A is high risk because it has high churn, declining owner concentration, repeated TODO comments, two recent failed builds, and long pull-request review time.” This explanation converts a model score into a usable engineering signal.

The framework also uses a debt-prioritization score. Debt probability alone is insufficient because a highly debt-prone low-impact component may not require immediate repayment. The priority score combines predicted debt probability, component criticality, change frequency, dependency centrality, test coverage, production incident linkage, and estimated repayment effort. This prioritization layer supports rational backlog decisions rather than creating an unranked list of debt warnings.

7. Experimental Protocol

A rigorous empirical evaluation of TD-DevPredict requires multi-project, temporal, and cross-project validation. The experimental protocol is designed to avoid common threats such as data leakage, label contamination, and overfitting to project-specific development practices.

The dataset should be constructed from open-source or enterprise repositories with sufficient history, issue-tracker data, pull-request records, and CI/CD logs. Candidate repositories must include at least one year of commit history, active issue management, code review metadata, and identifiable release or sprint windows. For open-source evaluation, repositories from Apache, Eclipse, Chromium, Kubernetes-related projects, and large Java or Python ecosystems may be used depending on data availability. For enterprise evaluation, anonymized internal repositories can be processed using the same feature extraction logic.

The evaluation unit should be component-time-window pairs. For example, each file or package is represented every two weeks or every sprint. Features are computed from historical data available before the window, and labels are derived from debt evidence appearing after the window. This temporal design is more realistic than random train-test splitting because continuous software development requires prediction on future unseen activity.

The baselines should include static-code-only models, churn-only models, SATD-only models, developer-activity-only models, and combined repository-mining models. TD-DevPredict is expected to outperform isolated baselines because it integrates multiple signal groups. However, the evaluation must test this hypothesis rather than assume it. Ablation studies should remove feature groups one at a time to measure the contribution of structural, change, textual, developer, and CI/CD features.

The metrics should include precision, recall, F1-score, area under the precision-recall curve, Matthews correlation coefficient, calibration error, and top-k inspection usefulness. Top-k usefulness is particularly important because teams often have limited refactoring capacity. A model that correctly ranks the top 10 most debt-prone components may be more useful than a model with marginally higher global accuracy. Similar effort-aware thinking has been used in just-in-time quality assurance, where inspection resources are focused on risky changes rather than all changes [6].

The protocol should include within-project validation, cross-project validation, and time-based rolling validation. Within-project validation tests whether the model can learn from a project's own history. Cross-project validation tests generalizability across repositories. Rolling validation tests whether the model remains stable as development practices evolve. Because technical debt is context-sensitive, cross-project performance may be lower than within-project performance, but transfer learning and feature normalization can improve robustness.

A qualitative validation step is also recommended. Experienced developers, maintainers, or technical leads should review a sample of high-risk predictions and classify them as actionable, partially actionable, or not useful. This step is important because technical debt is not only a statistical label; it is a maintainability concern that depends on team goals, business context, and architectural intent.

8. Analytical Findings and Expected Model Behavior

The proposed framework is expected to reveal several important patterns. First, change-process features will likely provide stronger predictive power than static metrics alone. Static complexity indicates maintainability burden, but high churn and repeated modification show that the burden is active. A complex but stable module may not require immediate repayment, while a moderately complex module with high churn and fragmented ownership may be a stronger debt candidate.

Second, developer activity features are expected to improve prediction by capturing knowledge distribution. Ownership dispersion, low reviewer familiarity, and contributor turnover may indicate that no single developer has sufficient context to manage architectural consistency. These signals can explain why debt persists even when code-level indicators appear moderate. This supports the broader view that technical debt is a socio-technical phenomenon rather than only a code-quality problem [24].

Third, SATD and textual features are expected to have high precision but incomplete recall. Explicit comments such as TODO and FIXME provide strong evidence when present, but many debt items are not self-admitted. Issue trackers and pull requests can improve coverage by capturing discussions that never appear in source code. Research on automatic SATD identification across multiple sources supports the value of combining comments, commit messages, pull requests, and issue trackers [27].

Fourth, CI/CD features are expected to be especially useful for identifying test debt, build debt, and operational debt. Repeated build failures, flaky tests, and deployment rollbacks may reveal debt that static analysis cannot detect. This is consistent with modern cloud and observability-centered engineering, where reliability depends on deployment pipelines, monitoring, and automated root-cause analysis rather than only source-code structure [29].

Fifth, explainability will determine whether teams adopt the model. A black-box technical debt score may be ignored by developers who need concrete evidence. Local explanations can translate model output into actionable recommendations, such as "increase test coverage," "assign stable ownership," "refactor duplicated logic," or "review recurring build failures." This is especially important in regulated or auditable environments where automated decisions must be understandable [9].

The practical outcome of TD-DevPredict is not only debt detection but debt governance. The model helps teams decide whether to repay debt immediately, schedule it for a future sprint, monitor it, or accept it as a deliberate trade-off. This aligns with agile lifecycle governance models where predictive analytics inform decision-making without replacing human judgment [7].

9. Practical Implementation in Continuous Software Development

TD-DevPredict can be implemented as a service integrated with Git platforms, CI/CD pipelines, issue trackers, and engineering dashboards. The service runs on a scheduled basis or after specific events such as pull-request creation, merge completion, build failure, or sprint closure. For each component, the system computes updated features, runs the prediction model, generates explanations, and stores results in a debt-risk dashboard.

At the pull-request level, the model can provide lightweight warnings. For example, if a pull request modifies a high-debt-risk component, the system can recommend additional reviewer attention, targeted test execution, or architectural review. The warning should not automatically block the merge unless the organization explicitly configures high-risk thresholds. The goal is to support informed engineering decisions, not create unnecessary process friction.

At the sprint-planning level, the model can rank components by debt probability and repayment priority. Engineering leads can allocate refactoring capacity based on predicted risk, business criticality, and team workload. This helps prevent a common agile failure mode where refactoring is repeatedly postponed because debt is invisible until it causes delivery delay.

At the release-readiness level, the model can identify debt clusters that may threaten production stability. For example, a release containing multiple changes to high-risk modules with recent build failures and low review depth may require additional regression testing. This connects technical debt prediction with release governance, deployment risk, and operational resilience.

The framework is also useful for enterprise modernization. Legacy systems often accumulate debt through undocumented dependencies, fragmented ownership, outdated testing practices, and manual deployment processes. Repository mining can identify unstable modules, while developer analytics can reveal knowledge concentration and onboarding risk. In such environments, technical debt prediction supports modernization roadmaps by identifying which components should be refactored, wrapped, migrated, or retired.

A cloud-native implementation can use containerized feature extraction workers, message queues for event processing, a feature store for time-windowed metrics, and a model-serving API. Static analysis results can be stored with repository metadata. CI/CD logs can be streamed into a data warehouse. Textual artifacts can be embedded using lightweight language models. Prediction outputs can be displayed in dashboards or added to issue trackers as recommended debt items.

Security and privacy must be considered. Developer analytics should not be used for individual surveillance or performance ranking. The purpose is to assess socio-technical risk at the component and team level, not to blame developers. The framework should anonymize or aggregate personal metrics where possible. This is particularly important in organizations that adopt predictive analytics across software lifecycle governance and cybersecurity-sensitive workflows [13].

10. Threats to Validity

Several threats must be addressed when evaluating TD-DevPredict. The first is construct validity. Technical debt is difficult to define, and labels derived from comments, issues, or refactoring commits may not capture all debt. Some debt is intentional and acceptable, while some high-complexity code is not debt. Hybrid labeling and expert validation reduce this threat but cannot eliminate it.

The second threat is data leakage. Repository-mining models can accidentally use future information, such as debt-removal commits or later issue comments, when predicting past debt. Temporal splitting and snapshot-based feature computation are required to ensure realistic evaluation.

The third threat is project bias. A model trained on one ecosystem may not generalize to another because languages, review practices, testing culture, and architecture styles differ. Cross-project validation and transfer learning experiments are necessary to evaluate generalizability.

The fourth threat is developer behavior sensitivity. Once developers know that TODO comments or review delays influence debt predictions, they may change documentation behavior. This does not necessarily invalidate the model, but it means the framework must evolve and include multiple signals rather than depending on a single easily manipulated indicator.

The fifth threat is organizational adoption. Even accurate predictions may fail if they are not integrated into planning routines. Technical debt tools must support existing engineering workflows, provide understandable explanations, and avoid excessive alerts. Otherwise, developers may treat the system as another dashboard that does not affect real decisions.

11. Discussion

The main argument of this paper is that technical debt prediction should move from static detection to continuous, explainable, socio-technical forecasting. Repository mining provides the historical evidence, developer analytics provides the human context, textual analysis provides explicit debt signals, and CI/CD data provides operational symptoms. Machine learning can integrate these signals into actionable predictions.

This approach also changes the role of technical debt management. Instead of asking teams to periodically inspect debt after velocity declines, TD-DevPredict enables early warning. It can identify components where debt is forming, explain the likely causes, and recommend targeted interventions. This is more aligned with continuous delivery because teams can act while the relevant code, context, and developers are still active.

The framework does not assume that all debt must be eliminated. Some technical debt is rational when business urgency requires rapid delivery. The real problem is unmanaged debt. A predictive framework allows teams to make explicit trade-offs:

accept low-risk debt, monitor medium-risk debt, and repay high-risk debt before it compounds. This distinction is critical for agile environments where perfect design is neither feasible nor economically optimal.

The proposed model also has relevance to AI-enabled software engineering. As machine learning and generative AI tools become more common in development workflows, new forms of debt may emerge through generated code, weak tests, poorly understood dependencies, and excessive automation trust. Prior work on hidden technical debt in machine learning systems shows that ML pipelines introduce debt through data dependencies, configuration complexity, and feedback loops [16]. Similar concerns apply when AI-assisted development accelerates code generation without equivalent improvements in review, testing, and maintainability governance.

12. Conclusion

This paper proposed TD-DevPredict, a machine learning-based framework for predicting technical debt in continuous software development using repository mining and developer activity analytics. The framework treats technical debt as a temporal socio-technical risk rather than a static code defect. It integrates structural code metrics, change-process signals, self-admitted technical debt evidence, issue and pull-request semantics, developer ownership analytics, and CI/CD stability features. The proposed methodology includes temporal labeling, explainable ensemble learning, debt-type classification, and repayment prioritization.

The paper contributes a research-oriented architecture that can be implemented in modern agile and DevOps environments. Its practical value lies in helping teams identify debt-prone components early, explain prediction drivers, allocate refactoring capacity, and connect technical debt management with sprint planning and release governance. Future work should empirically evaluate the framework across multiple open-source and enterprise repositories, compare structured and textual feature fusion strategies, explore graph neural networks for dependency-aware debt prediction, and develop human-centered dashboards that support responsible adoption without developer surveillance.

The long-term goal is to make technical debt visible before it becomes a delivery crisis. By combining repository mining with developer activity analytics, machine learning can help software teams move from reactive debt repayment to proactive maintainability governance.

References

[1] R. R. Thalakanti and S. S. Goud Bandari, “Intelligent Continuous Integration and Delivery for Banking Systems using Machine Learning Driven Risk Detection with Real World Deployment Evaluation,” *International Journal of AI, BigData, Computational and Management Studies*, vol. 5, no. 4, pp. 168–175, 2024, https://doi.org/10.63282/3050-9416.IJAIBDCMS-V5I4P118.

[2] A. Potdar and E. Shihab, “An Exploratory Study on Self-Admitted Technical Debt,” in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Victoria, BC, Canada, 2014, pp. 91–100, https://doi.org/10.1109/ICSME.2014.31.

[3] S. R. Gudi, “Design and Evaluation of Secure Microservices Architecture for HIPAA-Compliant Prescription Processing on AWS and OpenShift,” *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, vol. 5, no. 2, pp. 144–149, 2024, https://doi.org/10.63282/3050-9262.IJAIDSML-V5I2P116.

[4] Z. Li, P. Avgeriou, and P. Liang, “A Systematic Mapping Study on Technical Debt and Its Management,” *Journal of Systems and Software*, vol. 101, pp. 193–220, 2015, https://doi.org/10.1016/j.jss.2014.12.027.

[5] S. K. Gunda, S. D. R. Yettapu, S. Bodakunti, and S. B. Bikki, “Decision Intelligence Methodology for AI-Driven Agile Software Lifecycle Governance and Architecture-Centered Project Management,” 2023 Mar. 30, vol. 4, no. 1, pp. 102–108, https://doi.org/10.63282/3050-9262.IJAIDSML-V4I1P112.

[6] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A Large-Scale Empirical Study of Just-in-Time Quality Assurance,” *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013, https://doi.org/10.1109/TSE.2012.70.

[7] S. D. Sivva, “An End-to-End AI-Based Systems Engineering Paradigm for Lifecycle Governance, Predictive Quality Assurance, Automation Economics, and Cybersecurity Intelligence,” *Journal of Frontiers in Multidisciplinary Research*, vol. 4, no. 1, pp. 600–604, 2023, https://doi.org/10.54660/.JFMR.2023.4.1.600-604.

[8] R. Moser, W. Pedrycz, and G. Succi, “A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction,” in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, Leipzig, Germany, 2008, pp. 181–190, https://doi.org/10.1145/1368088.1368114.

[9] S. S. G. Bandari, S. D. Sivva, and R. R. Thalakanti, “Regulatory Grade Fraud Detection using Explainable Artificial Intelligence with Auditable Decision Pathways and Empirical Validation on Banking Data,” *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, vol. 5, no. 3, pp. 139–147, 2024, https://doi.org/10.63282/3050-9262.IJAIDSML-V5I3P115.

[10] E. da S. Maldonado and E. Shihab, “Detecting and Quantifying Different Types of Self-Admitted Technical Debt,” in *Proceedings of the 7th International Workshop on Managing Technical Debt (MTD)*, Bremen, Germany, 2015, pp. 9–15.

[11] S. R. Gudi, "Enhancing Reliability in Java Enterprise Systems through Comparative Analysis of Automated Testing Frameworks," *International Journal of Emerging Trends in Computer Science and Information Technology*, vol. 4, no. 2, pp. 151–160, 2023, https://doi.org/10.63282/3050-9246.IJETCSIT-V4I2P115.

[12] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," in *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, St. Louis, MO, USA, 2005, pp. 284–292, https://doi.org/10.1145/1062455.1062514.

[13] R. R. Thalakanti, S. S. Goud Bandari, and S. D. Sivva, "Federated Learning for Privacy Preserving Fraud Detection across Financial Institutions: Architecture Protocols and Operational Governance," *International Journal of Emerging Research in Engineering and Technology*, vol. 5, no. 2, pp. 108–114, 2024, https://doi.org/10.63282/3050-922X.IJERET-V5I2P111.

[14] F. Rahman and P. Devanbu, "Ownership, Experience and Defects: A Fine-Grained Study of Authorship," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, Honolulu, HI, USA, 2011, pp. 491–500, https://doi.org/10.1145/1985793.1985860.

[15] S. K. G. Gunda, "The Future of Software Development and the Expanding Role of ML Models," *International Journal of Emerging Research in Engineering and Technology*, vol. 4, no. 2, pp. 126–129, 2023, https://doi.org/10.63282/3050-922X.IJERET-V4I2P113.

[16] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden Technical Debt in Machine Learning Systems," in *Advances in Neural Information Processing Systems*, vol. 28, 2015, pp. 2503–2511.

[17] W. Cunningham, "The WyCash Portfolio Management System," in *Proceedings of the 7th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Vancouver, BC, Canada, 1992, pp. 29–30, https://doi.org/10.1145/157709.157715.

[18] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, "Identifying Self-Admitted Technical Debt in Open Source Projects Using Text Mining," *Empirical Software Engineering*, vol. 23, no. 1, pp. 418–451, 2018.

[19] S. R. Gudi, "Leveraging Predictive Analytics and Redis-Backed Caching to Optimize Specialty Medication Fulfillment and Pharmacy Inventory Management," *International Journal of AI, BigData, Computational and Management Studies*, vol. 5, no. 3, pp. 155–160, 2024, https://doi.org/10.63282/3050-9416.IJAIBDCMS-V5I3P116.

[20] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't Touch My Code! Examining the Effects of Ownership on Software Quality," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*, Szeged, Hungary, 2011, pp. 4–14, https://doi.org/10.1145/2025113.2025119.

[21] S. D. Sivva, R. R. Thalakanti, S. S. G. Bandari, and S. D. R. Yettapu, "AI-Driven Decision Intelligence for Agile Software Lifecycle Governance: An Architecture-Centered Framework Integrating Machine Learning Defect Prediction and Automated Testing," *International Journal of Emerging Trends in Computer Science and Information Technology*, vol. 4, no. 4, pp. 167–172, 2023, https://doi.org/10.63282/3050-9246.IJETCSIT-V4I4P118.

[22] Y. Li, M. Soliman, and P. Avgeriou, "Identifying Self-Admitted Technical Debt in Issue Tracking Systems Using Machine Learning," *Empirical Software Engineering*, vol. 27, article no. 131, 2022, https://doi.org/10.1007/s10664-022-10128-3.

[23] S. K. Gunda, "Comparative Analysis of Machine Learning Models for Software Defect Prediction," in *2024 International Conference on Power, Energy, Control and Transmission Systems (ICPECTS)*, Chennai, India, 2024, pp. 1–6, https://doi.org/10.1109/ICPECTS62210.2024.10780167.

[24] M. Balerao, "A Converged Artificial Intelligence Architecture for Innovation, Software Lifecycle Optimization, and Cybersecurity Risk Mitigation," *International Journal of Multidisciplinary Futuristic Development*, vol. 4, no. 1, pp. 117–120, 2023, https://doi.org/10.54660/IJMFD.2023.4.1.117-120.

[25] D. Tsoukalas, N. Mittas, A. Chatzigeorgiou, D. Kehagias, A. Ampatzoglou, T. Amanatidis, and L. Angelis, "Machine Learning for Technical Debt Identification," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1645–1662, 2023, https://doi.org/10.1109/TSE.2021.3124372.

[26] S. R. Gudi, "AI-Driven Fax-to-Digital Prescription Automation: A Cloud-Native Framework Using OCR, Machine Learning, and Microservices for Pharmacy Operations," *International Journal of Emerging Research in Engineering and Technology*, vol. 5, no. 1, pp. 111–116, 2024, https://doi.org/10.63282/3050-922X.IJERET-V5I1P113.

[27] Y. Li, M. Soliman, and P. Avgeriou, "Automatic Identification of Self-Admitted Technical Debt from Four Different Sources," *Empirical Software Engineering*, vol. 28, article no. 65, 2023, https://doi.org/10.1007/s10664-023-10297-9.

[28] S. K. Gunda, "Fault Prediction Unveiled: Analyzing the Effectiveness of Random Forest, Logistic Regression, and KNeighbors," in *2024 2nd International Conference on Self Sustainable Artificial Intelligence Systems (ICSSAS)*, Erode, India, 2024, pp. 107–113, https://doi.org/10.1109/ICSSAS64001.2024.10760620.

[29] I. Manga, S. D. Sivva, and V. K. Manga, "The Adaptive Intelligence in Cloud Systems: A Unified Architecture for AI Enhanced Observability and Automated Root Cause Analysis," *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, vol. 5, no. 1, pp. 160–166, 2024, https://ijaidsml.org/index.php/ijaidsml/article/view/366.

[30] N. Mutyam, "Graph-Based Modeling of Service Dependencies for Predicting Failure Propagation in Distributed Systems," *International Journal of Multidisciplinary Evolutionary Research*, vol. 5, no. 1, pp. 113–116, 2024, https://doi.org/10.54660/IJMER.2024.5.1.113-116.

[31] R. R. Thalakanti, S. S. Goud Bandari, and S. D. Sivva, "Federated Learning for Privacy Preserving Fraud Detection across Financial Institutions: Architecture Protocols and Operational Governance," *International Journal of Emerging Research in Engineering and Technology*, vol. 5, no. 2, pp. 108–114, 2024, https://doi.org/10.63282/3050-922X.IJERET-V5I2P111.

[32] S. D. Sivva, S. D. R. Yettapu, R. R. Thalakanti, and S. S. G. Bandari, "AI-Driven Decision Intelligence for Agile Software Lifecycle Governance: An Architecture-Centered Framework Integrating Machine Learning Defect Prediction and Automated Testing," *International Journal of Emerging Trends in Computer Science and Information Technology*, vol. 4, no. 4, pp. 167–172, 2023, https://doi.org/10.63282/3050-9246.IJETCSIT-V4I4P118.