ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may1527

A Literature-Grounded Holistic Sprint-Level Framework for Technical Debt Risk Prediction and Actionable Intervention in Agile Software Projects

Pranita Kumar[1]; Shripad Bhide[2]

[1]Post Graduate Student, Department of Computer Engineering
P.E.S. Modern College of Engineering Pune, India
[2]Assistant Professor, Department of Computer Engineering
P.E.S. Modern College of Engineering Pune, India

Publication Date: 2026/06/02

Abstract: Technical debt (TD) has become one of the most persistent challenges in modern agile software development. When development teams operate under tight sprint deadlines, they often make suboptimal design and implementation decisions that appear harmless in the short term but gradually erode software quality over time. Despite growing awareness of this problem, most existing approaches to managing technical debt remain reactive — teams typically address debt only after it has already accumulated, rather than identifying and preventing it early. This paper introduces a literature-grounded holistic sprint-level framework designed to predict technical debt risk and recommend targeted intervention strategies in agile software projects. The framework combines technical metrics — including code churn, cyclomatic complexity, defect density, velocity deviation, refactoring frequency, and maintainability index — with non-technical indicators such as team burnout score, documentation completeness, and sprint planning accuracy within a unified predictive architecture. The framework conceptually incorporates two interpretable machine learning approaches — Logistic Regression and Random Forest — to support sprint-level technical debt risk classification into Low, Moderate, and High categories. Each risk level is associated with intervention recommendations derived from peer-reviewed literature. This study is theoretical in scope and grounded in secondary empirical synthesis. No real-world data collection, coding, or experimental validation was conducted. The proposed framework is conceptually supported through systematic synthesis of empirical findings from nineteen peer-reviewed studies. Empirical implementation and validation using real-world sprint datasets are identified as important directions for future work. The primary contribution of this paper is a unified theoretical framework that integrates technical and non-technical factors, interpretable machine learning approaches, and risk-driven intervention strategies to support proactive technical debt governance in agile software development environments.

Keywords: *Technical Debt, Agile Software Development, Sprint-Level Prediction, Literature-Grounded Framework, Logistic Regression, Random Forest, Non-Technical Debt, Predictive Analytics, Actionable Intervention, Theoretical Framework, Secondary Empirical Synthesis.*

How to Cite: Pranita Kumar; Shripad Bhide (2026) A Literature-Grounded Holistic Sprint-Level Framework for Technical Debt Risk Prediction and Actionable Intervention in Agile Software Projects. *International Journal of Innovative Science and Research Technology*, 11(5), 2674-2684. https://doi.org/10.38124/ijisrt/26may1527

I. INTRODUCTION

Contemporary software teams face relentless pressure. They are expected to ship working software fast, respond to changing requirements quickly, and maintain high quality — all at the same time. Agile methodologies have become the standard way of managing this pressure, organizing work into short, focused iterations called sprints [7]. While agile practices improve development speed and flexibility, they can also introduce long-term maintenance and quality challenges. The same iterative pace that makes agile teams productive also creates an environment where technical debt quietly builds up, often going unnoticed until it becomes a serious problem.

Technical debt is the hidden price a team pays for taking shortcuts. When developers rush to meet a sprint deadline — skipping refactoring, cutting corners on documentation, or ignoring architectural warnings — they create debt that will cost more to fix later than it would have cost to do it right the first time [13]. Research has shown that software organizations spend roughly 25% of their total development

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may1527

effort dealing with TD-related problems [2]. That is a significant drain on time, money, and team morale.

What makes this especially difficult in agile environments is how naturally sprint pressure encourages debt accumulation. Developers postpone refactoring to hit delivery targets. They skip exception handling to stay on schedule. They compromise architectural decisions to close out stories before a sprint ends [12]. Practices like retrospectives and continuous integration help teams catch some of these issues, but they are fundamentally reactive tools — they identify debt after it has formed, not before [1]. In fast-moving agile projects, this is often too late.

There is growing evidence that the problem runs deeper than code. Research by Ahmad et al. showed that factors like team burnout, unclear roles, knowledge gaps, and poor sprint planning are just as responsible for technical debt as technical issues [2]. Similarly, Martini et al. found that social and process-level dysfunctions within agile teams can amplify technical debt in ways that pure code analysis cannot capture [14]. Yet, despite this understanding, most existing tools and frameworks focus primarily on the technical side. They analyze code metrics and source repositories but often overlook the human and organizational conditions that contribute to the problem.

This gap motivated the present work. This paper proposes a holistic sprint-level framework designed to predict technical debt risk before it accumulates and to associate each predicted risk level with specific, literature-grounded intervention strategies. The structure is based on two fundamental concepts. First, predicting TD risk requires considering both technical and non-technical signals within each sprint. Second, prediction alone is not sufficient — teams also require clear guidance on how to respond when risk is identified. By combining these elements, the framework aims to support a shift from reactive debt management toward proactive sprint-level governance.

➢ *The Main Contributions of this Study are:*

- Holistic Feature Integration: A unified sprint-level feature set that combines empirically validated technical metrics with non-technical human and process indicators for TD risk prediction.
- Sprint-Level Predictive Modelling: A structured conceptual prediction architecture incorporating Logistic Regression and Random Forest approaches to support sprint-level TD risk classification before debt becomes critical.
- Risk-Triggered Intervention Recommendations: A direct mapping from predicted risk levels to literature-grounded intervention strategies, providing agile teams with actionable guidance across different risk conditions.
- Holistic TD Governance Framework: A five-layer architecture integrating metric collection, feature engineering, conceptual ML-based risk prediction, risk classification, and intervention recommendations into a unified framework for proactive TD governance.

The organization of the remainder of the document is as follows.Section II reviews existing literature and identifies the research gap. Section III describes the proposed framework. Section IV presents the predictive modelling overview. Section V discusses conceptual validation. Section VI presents results and implications. Section VII wraps up the paper and highlights potential avenues for future research.

It is important to note that this study is theoretical in scope. The proposed framework is grounded in and conceptually supported through systematic synthesis of peer-reviewed literature rather than experimental implementation or empirical evaluation. This approach is consistent with established theoretical framework research in software engineering and provides a conceptual foundation for future empirical work. Practical implementation of the framework, including Python-based model development, real-world sprint data collection, and empirical model evaluation, is identified as an important direction for future research and will be addressed in subsequent work.

II. LITERATURE REVIEW

Over the past two decades, technical debt has been studied from many different angles — from how it forms and grows to how teams try to manage and prevent it. This section reviews the most relevant existing work across five key themes, gradually building toward the gap that this study addresses.

➢ *Technical Debt in Agile Environments*

The idea of technical debt was first introduced to explain what happens when developers take the easy route now and pay a higher price later [17]. Since then, researchers have worked to understand what kinds of debt exist and how they can be managed. Li et al. put together one of the most widely referenced taxonomies of technical debt, covering six major types: code debt, design debt, architectural debt, test debt, documentation debt, and process debt [13]. Their work also identified refactoring and architectural restructuring as the most common ways teams deal with accumulated debt.

Behutiye et al. took this a step further by looking specifically at technical debt in agile environments [7]. Through a systematic literature review, they found eight major causes of TD in agile projects. The most common were rapid delivery pressure, poor sprint estimation, and architectural shortcuts — which are all deeply tied to how agile teams work under time constraints. They also documented twelve different management strategies, with refactoring coming out on top. Despite how thorough this study is, it does not offer any way to predict when or where debt will form, and it does not look at individual sprint cycles specifically.

Leite et al. built on this work through a systematic mapping of 39 published articles on TD management in agile environments [1]. Their findings paint a clear picture: most existing research concentrates on identifying and monitoring debt, while early prevention and prediction remain largely ignored. This represents an important observation. If the

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may1527

research community is focused mostly on finding debt after it already exists, then there is still a major gap when it comes to catching it early — and this represents an important area addressed in the present study.

➢ *Non-Technical Dimensions of Technical Debt*

One of the more important developments in recent technical debt research is the recognition that debt is not solely a code-related problem. Martini et al. were among the first researchers to formally identify social debt and process debt as contributing factors that can intensify technical debt accumulation [14]. Their case study in large-scale agile environments showed that poor inter-team communication and misaligned processes can accelerate technical debt accumulation — a finding with important implications for predictive frameworks.

Ahmad et al. extended this perspective by conducting in-depth interviews with 24 practitioners across four large international organizations [2]. Their findings identified five distinct non-technical contributors to technical debt: social dynamics, process-related issues, people-related challenges, documentation gaps, and requirements instability. Specific factors such as knowledge silos, unclear roles, scope creep, and team burnout were highlighted as important contributors. This work strongly supports the inclusion of non-technical indicators in the proposed framework. However, because these studies are primarily qualitative in nature, they do not provide quantitative sprint-level prediction models.

➢ *Industrial Practice and Measurement*

Beyond research literature, it is also important to understand how technical debt is managed in industrial practice. Codabux and Williams studied Scrum teams and found that prioritization of technical debt is often handled informally, based on perceived severity or customer impact [8]. Design debt, testing debt, and defect-related debt were identified as the most frequently encountered forms. The absence of a structured prioritization mechanism represents a practical limitation that data-driven approaches may help address.

Sneed examined how debt accumulates across sprint cycles and observed that developers frequently skip refactoring and quality checks under deadline pressure [12]. His study also identified static code analysis and cost estimation models as practical mechanisms for measuring debt levels. Fowler, whose work on refactoring remains foundational in software engineering, described code smells as indicators of deeper structural issues within software systems [19]. Collectively, these studies support the importance of measurement and early detection, although they do not provide mechanisms for proactive sprint-level risk prediction.

➢ *TD Prevention and Intervention*

In the area of technical debt prevention, Freire et al. conducted one of the most comprehensive studies currently available [9]. They identified 136 prevention practices organized across seven categories: project planning, good practices, team quality, testing, requirements management, process improvement, and TD management. This catalogue directly informs the intervention strategies incorporated within the proposed framework.

In a follow-up study, the same research group investigated why teams often fail to implement prevention practices even when they are aware of them [10]. The primary barriers included organizational resistance, time pressure, and the absence of automated mechanisms capable of indicating when intervention is necessary. This finding is especially relevant to the present study because it suggests that awareness alone is insufficient — teams also require timely signals indicating when corrective action should be taken. The proposed framework is intended to support this need.

➢ *AI/ML Approaches and Predictive Analytics*

The use of artificial intelligence in technical debt management has expanded significantly in recent years. Alam Binta et al. reviewed a range of AI-based approaches including natural language processing, static code analysis, and deep learning tools such as SonarQube and ARCAN [3]. Although these approaches demonstrate promising capabilities, they remain primarily focused on source code analysis and do not incorporate sprint-level process indicators or intervention guidance mechanisms.

Tsintzira et al. identified sprint-level machine learning prediction of TD as an important direction for future research [15]. Mathioudaki et al. demonstrated that LSTM-based deep learning models can forecast future technical debt values with reasonable accuracy, although their approach operates at the project level and presents interpretability challenges [16].

In the broader context of machine learning within agile environments, Hanslo and Tanner applied Multiple Linear Regression to predict Scrum adoption and found sprint management and project complexity to be statistically significant predictors [4]. ForouzeshNejad et al. reported strong predictive performance using an Artificial Neural Network combined with SHAP explainability techniques for agile project success prediction [5]. While technically sophisticated, their work focuses on overall project success rather than technical debt prediction and does not explicitly incorporate non-technical sprint-level indicators. Azonuche and Enyejo identified sprint velocity, defect rates, and code churn as meaningful agile risk indicators [6], although they did not develop a dedicated framework for technical debt prediction.

➢ *Research Gap*

Reviewing the existing literature reveals a recurring research gap. As shown in Table I, limited existing work jointly integrates sprint-level analysis, technical and non-technical indicators, interpretable machine learning approaches, technical debt prediction, and actionable intervention recommendations within a unified framework. Most prior studies address only selected dimensions while leaving others insufficiently explored.

This gap is important because effective technical debt governance in agile environments requires these dimensions

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may1527

to operate together. Identifying debt after it has accumulated is often insufficient — teams also require mechanisms capable of estimating when debt risk is increasing, understanding contributing factors, and recommending appropriate interventions. The framework proposed in this study is designed to support these objectives through an integrated sprint-level perspective.

Table 1 Comparison of Existing Works

Reference	Sprint-Level	Non-Technical	ML Model	TD Prediction	Intervention
Leite et al. [1]	✗	✗	✗	✗	✗
Ahmad et al. [2]	✗	✓	✗	✗	✗
Alam Binta et al. [3]	✗	✗	✓	✓ Partial	✗
Hanslo & Tanner [4]	✗	✗	✓	✗	✗
ForouzeshNejad et al. [5]	✓ Partial	✗	✓	✓ Partial	✗
Azonuche & Enyejo [6]	✗	✗	✗	✓ Partial	✗
Behutiye et al. [7]	✗	✗	✗	✗	✗
Codabux & Williams [8]	✗	✗	✗	✗	✗
Freire et al. [9]	✗	✗	✗	✗	✓ Partial
Freire et al. [10]	✗	✗	✗	✗	✗
Domingues et al. [11]	✓ Partial	✗	✗	✗	✗
Sneed [12]	✗	✗	✗	✗	✗
Mathioudaki et al. [16]	✗	✗	✓	✓	✗
Proposed Framework	✓	✓	✓	✓	✓

III. RESEARCH METHODOLOGY

➢ *Research Approach*

This study does not involve coding, data collection, or experimental evaluation. Instead, it is built on findings synthesized from nineteen peer-reviewed studies to design a conceptual framework for predicting technical debt risk at the sprint level.

This type of research follows a secondary empirical synthesis approach, where findings from existing studies are analyzed and integrated to construct a new conceptual framework. This is a well-established research approach in software engineering. Several widely cited studies in this domain, including Behutiye et al. [7] and Li et al. [13], were also developed through literature-based synthesis.

The suitability of this approach stems from the availability of prior research identifying sprint metrics related to technical debt, machine learning approaches suitable for software engineering prediction tasks, and intervention strategies for agile environments. Rather than reproducing existing empirical findings, this study integrates evidence from prior literature into a unified conceptual framework for proactive technical debt governance.

➢ *Research Design*

The study was carried out in three clear phases. Every stage relied on the one before it.

- *Phase 1 — Reading and Understanding Existing Research:*

The first thing done was a thorough review of existing literature across five key areas — what technical debt is and how it forms in agile environments, the human and organizational side of debt, how real industry teams currently manage debt, what prevention actions exist, and how machine learning has been applied to software quality problems. Nineteen studies were selected from trusted academic sources including IEEE, ACM, Elsevier, Springer, and MDPI. Only studies directly relevant to sprint-level technical debt prediction and agile software engineering were included.

- *Phase 2 — Designing the Framework:*

Once the literature was reviewed and understood, the framework was designed. Framework design decisions made during this phase — including metric selection, modelling approaches, risk classification, and intervention mapping — were informed by evidence reported in the reviewed literature. Nothing was assumed or made up. If a metric was included, it was because at least one published study proved it relates to technical debt. If a model was selected, it was because established research confirmed its suitability for this type of classification problem.

It is also important to clarify the role of the mathematical formulation presented later in this paper. The formulas describing Logistic Regression, Random Forest, and evaluation metrics are included to show the theoretical structure of the proposed models — not to report experimental results. No model was actually trained, tested, or evaluated on any dataset. The mathematical formulation simply explains how these models would work if implemented in future empirical work.

- *Phase 3 — Validating the Framework:*

Since no experiments were conducted, validation was done conceptually. This means going back through the literature and showing, step by step, that every part of the framework is supported by existing empirical findings. This

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may1527

form of validation is standard practice in theoretical framework research and has been widely used in software engineering studies.

➢ *Scope and Limitations*

This paper is honest about what it is and what it is not. It is a theoretical framework paper. There is no code written, no dataset used, no model actually trained, and no accuracy numbers calculated. The mathematical formulas presented in this study describe the theoretical structure of the proposed models only. The framework tells you what should be built and why it would work — but the actual building is left for future work.

This is not a weakness — it is a deliberate and honest research choice. A well-designed theoretical framework gives future researchers and practitioners a clear, evidence-based blueprint to work from. Without such a foundation, implementation efforts often lack direction or justification. This paper provides that foundation.

That said, the most important limitation is clear: the framework has not yet been tested on real sprint data. Its practical performance — how accurate it is, how useful teams find it, how well it generalizes across different organizations — remains unknown until empirical work is carried out. Future work will focus on Python-based implementation, model training on real sprint datasets, and full experimental evaluation of the framework.

➢ *Ethical Considerations*

Since no human participants were involved and no real organizational data was collected, this study carries very low ethical risk. Every paper cited has been properly referenced and paraphrased — nothing has been copied directly. No data was fabricated or misrepresented at any point.

One thing worth mentioning: AI-assisted writing tools were used to help with language refinement during the writing process. However, every idea in this paper — the framework design, the metric selection, the model choices, the mathematical formulations, the intervention mapping, and all analytical conclusions — came entirely from the author. The AI helped with how things were said, not with what was said.

Finally, it is worth repeating clearly: this study does not claim to have built or tested anything experimentally. It proposes a theoretical framework grounded in nineteen peer-reviewed studies. Turning that proposal into a working, tested, and coded system is the next step — and it is identified as the top priority for future research.

IV. PROPOSED FRAMEWORK

The goal of this framework is straightforward: give agile teams a way to see technical debt risk coming before it becomes a real problem, and tell them exactly what to do about it. Most current approaches wait until debt has already built up before taking action. This framework addresses this limitation — it works at the sprint level, treating each sprint as its own unit of analysis, and uses both technical and human factors to predict where risk is heading.

The structure is arranged in five tiers. Each layer handles one part of the process, and together they form a complete pipeline from data collection to intervention recommendation.

➢ *Layer 1 — Holistic Metrics Consolidation Layer*

The first layer is about deciding what to measure. Feature selection is not arbitrary — every metric included here has been validated in prior research as a meaningful indicator of technical debt risk.

- *Technical Metrics:*

✓ Code Churn per Sprint: This measures how much code was added, deleted, or changed during a sprint. A sprint with very high churn is often a sign of unstable development — requirements changing too fast, or developers rushing through implementation. Research consistently links high churn to increased defect rates and debt accumulation [12,7].
✓ Cyclomatic Complexity: This tells us how structurally complex the code is. As complexity increases, the code becomes more challenging to read, test, and maintain.Over time this leads to deeper and more expensive technical debt [7].
✓ Defect Density per Sprint: The number of defects found relative to the size of the codebase. High defect density usually points to weak testing and poor quality control — both of which are closely linked to technical debt [3,8].
✓ Sprint Velocity Deviation: This is the gap between what a team planned to deliver in a sprint and what they actually finished. Large gaps often mean the team was either overcommitted or ran into unexpected issues — either way, shortcuts tend to follow [4,12].
✓ Refactoring Frequency: How often is the team improving the code without changing what it does? Low refactoring frequency is a warning sign — it means quality improvements are being put off, and debt is quietly building [9,19].
✓ Maintainability Index: A composite score that combines complexity, code size, and documentation quality. When this number drops, it usually means the codebase is getting harder to work with — a reliable early signal of technical debt [7,13].

- *Non-Technical Metrics:*

✓ Team Burnout Score: This measures the workload pressure on the team during a sprint. High burnout levels often lead to rushed development and lower code quality [2,14].
✓ Documentation Completeness Rate: This indicates how well the code and system components are documented. Poor documentation is a known contributor to technical debt [2,10].
✓ Sprint Planning Accuracy: This reflects how well planned sprint goals match the actual outcomes. Poor planning

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may1527

leads to process inefficiencies and development shortcuts [2,14].

✓ Code Review Coverage: This measures the proportion of code that has been reviewed by peers. Low review coverage allows defects and quality issues to remain undetected [9,14].

➢ *Layer 2 — Feature Engineering and Normalization Layer*

Once the data is collected, it needs to be prepared before any modelling can happen. Raw sprint data comes from different tools and sources and will not always be in a consistent format. This layer handles missing values, applies min-max normalization so that all metrics are on a comparable scale, and aggregates everything at the sprint level so each sprint becomes one complete data record. Only features validated in literature are kept — this supports interpretability and reduces the inclusion of irrelevant features.

➢ *Layer 3 — Predictive Modelling Layer*

This layer conceptually describes the prediction process. Two models are used rather than one, because each brings something different to the table.

- Logistic Regression works as the interpretable baseline. It conceptually estimates the probability of TD risk for each sprint and expresses its reasoning through coefficients — one for each metric. This makes it easy for a project manager or developer to look at the model output and understand exactly which factors are driving the risk score. Transparency matters in agile teams, where stakeholders need to trust the tools they use [18].
- Random Forest handles the more complex patterns. Real sprint data does not always follow simple linear rules. Sometimes it is the combination of several slightly-off metrics — moderate churn, slightly low refactoring, and a stressed team — that signals real danger. Random Forest is built to catch those kinds of interactions [17].

Using both models together allows the framework to support both interpretability and complex pattern identification (through LR) and accurately detect when it is risky even in complex situations (through RF).

➢ *Layer 4 — Sprint-Level Risk Classification Layer*

The conceptual prediction outputs from the proposed approaches are categorized into three risk levels that any team member can understand at a glance:

- Low Risk: Things are under control. Metrics are stable, and there are no major warning signs.
- Moderate Risk: Something is off. Early indicators suggest that debt could start building if nothing changes.
- High Risk: Multiple things are going wrong at once. Without intervention, significant debt accumulation is very likely.

➢ *Layer 5 — Intervention Recommendation Layer*

In the final layer, the predicted risk levels are directly linked to specific intervention strategies based on existing research findings. This allows the framework to conceptually associate predicted risk levels with clear guidance on what actions should be taken.

- *Risk Level to Intervention Mapping:*

✓ *Low Risk:*

- *Technical:* Continue current development practices and monitor key metrics.
- *Non-Technical:* Maintain team morale and ensure documentation is consistently updated.

✓ *Moderate Risk:*

- *Technical:* Schedule refactoring activities and increase code review efforts.
- *Non-Technical:* Re-evaluate sprint planning accuracy and address signs of team burnout.

✓ *High Risk:*

- *Technical:* Perform immediate refactoring, conduct architectural reviews, and reprioritize the sprint backlog.
- *Non-Technical:* Adjust team workload, organize documentation-focused sprints, and encourage knowledge sharing among team members.

V. MATHEMATICAL FORMULATION

➢ *Feature Vector Representation*

Each sprint is represented as a feature vector:

$$X=[x1,x2,x3,x4,x5,x6,x7,x8,x9,x10]$$

Where the variables represent sprint-level indicators including code churn, cyclomatic complexity, defect density, sprint velocity deviation, refactoring frequency, maintainability index, team burnout score, documentation completeness, sprint planning accuracy, and code review coverage.

The target variable is represented as:

$$Y\in\{0,1,2\}$$

Where:

- 0 represents Low Risk,
- 1 represents Moderate Risk,
- 2 represents High Risk.

This representation provides a structured foundation for conceptual sprint-level technical debt risk modelling.

➢ *Logistic Regression Model*

Logistic Regression is incorporated as an interpretable baseline approach for estimating sprint-level technical debt risk probabilities. The model conceptually estimates how individual sprint metrics contribute to predicted risk levels.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may1527

$P(Y = 1 \mid X) = 1 / (1 + e^{-(\beta_0 + \Sigma_{i=1}^{n} \beta_i x_i)})$

Where β0 is the intercept and βi are coefficients corresponding to each feature xi. These coefficients indicate how strongly each metric influences the predicted technical debt risk, making the model interpretable and suitable for decision-making.

➢ *Random Forest Model*

Random Forest is incorporated as a complementary ensemble learning approach intended to capture more complex relationships among sprint-level indicators. Unlike Logistic Regression, Random Forest can conceptually model nonlinear interactions between technical and non-technical factors that may collectively contribute to technical debt risk.

The combination of Logistic Regression and Random Forest supports a balance between interpretability and complex pattern identification within the proposed framework.

VI. CONCEPTUAL VALIDATION

This section validates the proposed framework by using a systematic synthesis of empirical findings reported in existing literature. Since this study follows a secondary empirical synthesis approach, validation is carried out by showing that each selected metric, model, and intervention strategy is supported by established research in software engineering.

➢ *Validation of Technical Metrics*

- Code Churn: Sneed observed directly that developers who rush through sprint work produce code changes that are inconsistent and error-prone [12]. Azonuche and Enyejo also flagged code churn as one of the most reliable early indicators of risk in agile projects [6]. The evidence is consistent across multiple studies.
- Cyclomatic Complexity: Behutiye et al. showed that as code complexity grows in agile projects, it becomes progressively harder to test and maintain [7]. This creates a compounding problem — complexity makes it harder to pay off existing debt, which allows even more debt to accumulate.
- Defect Density: Codabux and Williams found in their industry study that high defect density was the clearest observable sign of poor quality control in Scrum teams [8]. When defects pile up sprint after sprint, it is almost always a sign that something deeper is wrong.
- Sprint Velocity Deviation: Hanslo and Tanner demonstrated a clear statistical link between how well sprints are managed and overall project outcomes [4]. When teams consistently miss their sprint targets, it creates pressure that almost inevitably leads to shortcuts and deferred quality work [12].
- Refactoring Frequency: Fowler's foundational work established that regular refactoring is the most reliable way to keep technical debt from accumulating [19]. Freire et al. confirmed the flip side of this — when refactoring is skipped or delayed, debt grows faster [9].
- Maintainability Index: Li et al. validated the maintainability index as one of the most reliable composite indicators of technical debt across multiple types of software projects [13]. When this score drops, the cost of future changes goes up.

➢ *Validation of Non-Technical Metrics*

- *Team Burnout Score:*

Non-technical factors play a major role in technical debt. Ahmad et al. identified team workload and burnout as key contributors to technical debt, as overworked developers tend to rush their work and skip quality practices [2]. Martini et al. also confirmed that burnout acts as a process-level driver of technical debt in large agile teams [14].

- *Documentation Completeness Rate:*

Documentation is essential for maintaining software quality. Ahmad et al. identified documentation debt as one of the major non-technical contributors to technical debt [2]. Poor documentation leads to knowledge gaps and makes maintenance more difficult. Freire et al. also highlighted that lack of proper documentation is a major barrier to effective technical debt prevention [10].

- *Sprint Planning Accuracy:*

Sprint planning accuracy reflects how well planned tasks match actual outcomes. Ahmad et al. showed that poor planning leads to process inefficiencies and forces developers to take shortcuts [2]. Sneed further observed that inaccurate planning under time pressure directly leads to quality compromises and technical debt accumulation [12].

- *Code Review Coverage:*

Code review is an important practice for maintaining code quality. Freire et al. identified code review as one of the most effective preventive actions against technical debt [9]. When review coverage is low, defects and design issues can go unnoticed, increasing the risk of technical debt over time.

➢ *Validation of Model Selection*

- *Logistic Regression:*

Logistic Regression is widely used for risk prediction because of its simplicity and interpretability. Hosmer and Lemeshow established it as a standard method for probabilistic classification [18]. In software engineering, it is commonly used for defect prediction and risk analysis. Its ability to provide clear coefficients makes it especially useful in agile environments where transparency and understanding are important.

- *Random Forest:*

Random Forest is known for its ability to capture complex relationships between variables. Breiman showed that it performs well in high-dimensional datasets and can model nonlinear interactions effectively [17]. ForouzeshNejad et al. also showed that ensemble methods

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may1527

can improve predictive performance in agile project analysis [5]. Mathioudaki et al. further supported the use of such models for technical debt forecasting [16].

Using both approaches together supports a balance between interpretability and complex pattern identification.

➢ *Validation of Intervention Recommendations*

Every intervention recommended by this framework traces back to established research. The refactoring-based interventions for High Risk sprints are grounded in Fowler's refactoring principles [19]. The backlog reprioritization recommendations reflect practices documented by Leite et al. for Scrum-based TD management [1]. Non-technical interventions like workload rebalancing and knowledge sharing are directly supported by Ahmad et al.'s findings on the human drivers of technical debt [2]. And the overall prevention action catalogue draws from Freire et al.'s comprehensive study of 136 TD prevention practices [9].

➢ *Integrated Framework Validation*

Taken together, the five layers of this framework form a coherent and evidence-based system. The integration of technical and non-technical metrics is supported by Ahmad et al. [2] and Martini et al. [14], both of whom established that these two dimensions of debt are deeply connected. The choice to work at the sprint level is supported by Domingues et al., who showed that sprint-by-sprint analysis gives more useful and actionable insights than project-level assessment [11]. No single existing study combines all of these elements — but together they validate every design decision made in this framework.

VII. RESULTS AND DISCUSSION

Since this study does not use a live dataset, the results section takes a different approach. Rather than reporting accuracy numbers, we walk through what the framework is expected to do in practice — using scenario analysis to show how the models would behave in real sprint situations, and drawing on literature to explain why those behaviors make sense.

➢ *Expected Model Behavior*

Based on what prior research tells us about the relationship between sprint metrics and technical debt, the Logistic Regression model is expected to produce the following coefficient directions:

- Code churn ($\beta_1 > 0$): More changes, more risk [12]
- Cyclomatic complexity ($\beta_2 > 0$): More complexity, more risk [7]
- Defect density ($\beta_3 > 0$): More defects, more risk [8]
- Velocity deviation ($\beta_4 > 0$): Bigger gaps, more risk [4]
- Refactoring frequency ($\beta_5 < 0$): More refactoring, less risk [9,19]
- Maintainability index ($\beta_6 < 0$): Higher score, less risk [13]
- Team burnout score ($\beta_7 > 0$): More burnout, more risk [2,14]
- Documentation completeness ($\beta_8 < 0$): Better docs, less risk [2,10]
- Sprint planning accuracy ($\beta_9 < 0$): Better planning, less risk [2]
- Code review coverage ($\beta_{10} < 0$): More reviews, less risk [9]

The Random Forest model is expected to go beyond these individual relationships and pick up on combinations that the linear model might miss. For example: a sprint with moderate code churn and moderate burnout might look manageable individually, but together with low refactoring frequency, they could signal serious risk.

➢ *Scenario-Based Analysis*

- Scenario 1 — High Risk Sprint: Imagine a sprint where 450 lines of code were modified, the team found 3.2 defects per thousand lines of code, refactoring happened only once, the team delivered 25% less than planned, documentation coverage was at 45%, and the team burnout score was 7 out of 10.

This sprint would be flagged as High Risk by both models. The LR model would show strong positive weights on churn, defects, and burnout. The RF model would pick up on the fact that all these factors are happening at the same time, which makes things worse than any single factor alone. Recommended actions: immediate refactoring, an architectural review, backlog reprioritization, and workload rebalancing for the team.

- Scenario 2 — Moderate Risk Sprint: A sprint where velocity deviation was 12%, defect density was moderate at 1.8 defects per KLOC, and documentation had some gaps — but nothing catastrophic.

Both models would classify this as Moderate Risk. It is not an emergency, but it is a warning. Recommended actions: schedule refactoring for the next sprint, increase code review participation, and look closely at sprint planning before the next cycle.

- Scenario 3 — Low Risk Sprint: A sprint with controlled code changes, low defect rates, stable velocity, regular refactoring, full documentation, and a well-rested team.

Both models would agree: Low Risk. The recommendation here is simple — keep doing what you are doing, and keep monitoring the metrics so things do not slip.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may1527

➢ *Comparative Model Analysis*

Table 2 Comparative Model Analysis

Criterion	Logistic Regression	Random Forest
Interpretability	High — easy to understand coefficients	Moderate — uses feature importance
Nonlinear Capture	Limited	Strong
Overfitting Resistance	Moderate	High due to ensemble learning
Stakeholder Transparency	High	Moderate
Recommended Use	Explaining why risk occurs	Identifying when risk occurs

This comparison shows that both models have their own advantages. Logistic Regression helps in understanding the reason behind risk, while Random Forest supports identification of complex sprint-level relationships. Using both together provides a balanced and effective solution.

➢ *Discussion*

What this framework ultimately offers is a shift in mindset. Right now, most agile teams manage technical debt the way a doctor treats symptoms instead of preventing illness. This framework is closer to a health monitoring system — it watches for warning signs before the problem becomes serious.

The inclusion of non-technical metrics is arguably the most important contribution. Existing ML-based TD tools focus entirely on code. But as Ahmad et al. showed, some of the biggest drivers of technical debt have nothing to do with code — they have to do with people [2]. A burned-out team, a poorly planned sprint, or a knowledge gap between developers can cause just as much debt as a complex codebase. Ignoring these factors means missing half the picture.

The choice of interpretable models also matters more than it might seem. A neural network might give slightly better accuracy, but if a developer cannot understand why a sprint was flagged as high risk, they are less likely to act on it. ForouzeshNejad et al. acknowledged this limitation in their own work [5]. The LR and RF combination in this framework balances accuracy with transparency in a way that should make adoption more realistic in real agile teams.

And finally, the intervention layer closes the loop. Freire et al. found that teams often fail to prevent TD not because they do not know what to do, but because there is no clear signal telling them when to do it [10]. This framework provides that signal.

VIII. IMPLICATIONS

➢ *Implications for Industry*

For agile teams and project managers, this framework offers something that most existing tools do not: a way to get ahead of technical debt instead of chasing it. By catching risk signals at the sprint level, teams can make smarter decisions about backlog priorities and how to allocate effort before problems become expensive.

The non-technical metrics are especially valuable from a management perspective. Most technical debt tools give developers code-level feedback. This framework also gives project managers visibility into team health, planning quality, and documentation standards — factors they can actually influence directly.

The interpretability of the models matters for adoption. If a team is told "this sprint is high risk" but cannot see why, they may ignore the warning. When the reasoning is transparent — "your code churn is high, your refactoring rate is low, and your team burnout score is elevated" — it is much easier to take the right action. And by providing specific recommended actions at each risk level, the framework reduces the guesswork that Codabux and Williams observed as a major weakness in current industry practice [8].

➢ *Implications for Research*

From a research perspective, this work opens up several new directions. The holistic approach — combining technical and non-technical indicators in a single predictive model — remains comparatively underexplored in the TD prediction literature.Future studies could test this approach empirically and refine the feature set based on real-world data.

The use of interpretable machine learning for TD prediction also connects to the growing field of explainable AI in software engineering. The framework highlights the importance of balancing interpretability with complex pattern identification.That balance is worth investigating further as AI tools become more common in software development workflows.

IX. CONCLUSION AND FUTURE WORK

➢ *Conclusion*

This paper set out to answer a practical question: can we predict technical debt risk at the sprint level before it becomes a serious problem, and can we tell teams what to do about it? The framework proposed here is our answer to that question.

By combining six technical metrics with four non-technical human and process indicators, the framework captures both technical and non-technical sprint-level indicators within a unified framework. Two interpretable machine learning approaches — Logistic Regression and Random Forest — are incorporated to support sprint-level risk classification into Low, Moderate, or High categories. And each risk level triggers specific intervention recommendations drawn from nineteen peer-reviewed studies.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may1527

The five-layer architecture ties all of this together into a coherent system — from data collection through risk classification to actionable output. Every component of the framework is validated by existing research, even if the framework as a whole has not yet been tested on real-world data.

What this work contributes most is a new way of thinking about technical debt in agile environments. TD is not just a code problem. It is a product of how teams work, how sprints are planned, and how people are treated. A framework that ignores the human side of the equation will always miss something important. This one does not.

This study is based entirely on secondary empirical synthesis of peer-reviewed literature. No primary data collection or experimental evaluation was conducted. Empirical validation using real-world sprint data is the most important next step.

➢ *Future Work*

While this study makes a meaningful theoretical contribution, several important directions remain open for future research.

The most immediate and critical next step is empirical implementation and experimental validation. Future work will involve developing a complete Python-based implementation of the proposed framework, including data preprocessing pipelines, Logistic Regression and Random Forest model training, hyperparameter tuning, and performance evaluation using standard classification metrics such as accuracy, precision, recall, and F1-score. This implementation will first be tested on synthetic sprint datasets generated based on empirically validated metric ranges from literature, followed by validation on real-world agile sprint data from sources such as JIRA project histories, GitHub repository metrics, and open-source software engineering datasets available through repositories such as the PROMISE data repository.

Beyond initial implementation, future research directions include:

- Model Comparison and Extension: The framework currently proposes Logistic Regression and Random Forest as predictive models. Future studies should compare these against more advanced algorithms such as XGBoost, gradient boosting, support vector machines, and explainable neural networks to identify the most effective approach for sprint-level TD risk prediction.
- DevOps and CI/CD Integration: Integrating the framework into DevOps pipelines and CI/CD platforms such as Jenkins, GitHub Actions, or Azure DevOps would enable real-time sprint-level TD risk monitoring and automatic triggering of intervention recommendations during sprint execution — transforming the framework from a theoretical tool into a practical operational system.
- Longitudinal Empirical Studies: Long-term studies tracking technical debt trends across multiple sprint cycles in organizations using the framework would provide strong empirical evidence of its practical impact on software quality and team productivity.
- Non-Technical Feature Expansion: Additional non-technical factors such as psychological safety scores, inter-team communication quality, and team dependency metrics could be incorporated to further improve the framework's practical applicability and predictive capability, building on the findings of Ahmad et al. [2].
- Tool Development: Development of a lightweight software tool or dashboard that automates metric collection, risk classification, and intervention recommendation generation would significantly improve the framework's practical adoption in real-world agile environments.

These directions collectively represent a comprehensive roadmap for transforming the theoretical framework presented in this paper into a fully validated, empirically tested, and practically deployable system for proactive technical debt governance in agile software development.

REFERENCES

[1]. G. de Souza Leite, R. E. P. Vieira, L. Cerqueira, R. S. P. Maciel, S. Freire, and M. Mendonça, "Technical Debt Management in Agile Software Development: A Systematic Mapping Study," in *Proc. XXIII Brazilian Symposium on Software Quality (SBQS 2024)*, Salvador, Brazil, Nov. 2024, pp. 1–12.

[2]. M. O. Ahmad, V. Mandić, N. Taušan, A. Katin, and P. Herath, "Technical debt is not just technical: An industrial case study in large agile software development," *Journal of Systems and Software*, vol. 234, p. 112719, Jan. 2026.

[3]. S. A. Binta, S. Kaushal, and S. B. Pandi, "Artificial Intelligence for Technical Debt Management in Software Development," Department of Software Engineering, LUT University, Lappeenranta, Finland, Student Research Paper, 2023.

[4]. R. Hanslo and M. Tanner, "Machine Learning Models to Predict Agile Methodology Adoption," in *Proc. 15th Federated Conference on Computer Science and Information Systems (FedCSIS)*, Sofia, Bulgaria, 2020, pp. 697–704.

[5]. A. A. ForouzeshNejad, F. Arabikhan, A. Gegov, R. Jafari, and A. Ichtev, "Data-Driven Predictive Modelling of Agile Projects Using Explainable Artificial Intelligence," *Electronics*, vol. 14, no. 13, p. 2609, Jun. 2025.

[6]. T. I. Azonuche and J. O. Enyejo, "Adaptive Risk Management in Agile Projects Using Predictive Analytics and Real-Time Velocity Data Visualization Dashboard," *International Journal of Innovative Science and Research Technology*, vol. 10, no. 4, pp. 2032–2047, Apr. 2025.

[7]. W. N. Behutiye, P. Rodríguez, M. Oivo, and A. Tosun, "Analyzing the concept of technical debt in the context of agile software development: A systematic literature review," *Information and Software Technology*, vol. 82, pp. 139–158, Feb. 2017.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may1527

[8]. Z. Codabux and B. Williams, "Managing Technical Debt: An Industrial Case Study," in *Proc. 4th International Workshop on Managing Technical Debt (MTD)*, San Francisco, CA, USA, May 2013, pp. 8–15.

[9]. S. Freire, A. Pacheco, N. Rios, B. Pérez, C. Castellanos, D. Correal, R. Ramač, V. Mandić, N. Taušan, G. López, M. Mendonça, D. Falessi, C. Izurieta, C. Seaman, and R. Spínola, "A Comprehensive View on TD Prevention Practices and Reasons for Not Preventing It," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, Art. 178, Sep. 2024.

[10]. S. Freire, N. Rios, B. Gutierrez, D. Torres, M. Mendonça, C. Izurieta, C. Seaman, and R. Spínola, "Surveying Software Practitioners on Technical Debt Payment Practices and Reasons for not Paying off Debt Items," in *Proc. International Conference on Evaluation and Assessment in Software Engineering (EASE 2020)*, Trondheim, Norway, Apr. 2020, pp. 210–219.

[11]. P. Domingues, M. Goulão, and J. Ferreira, "Tracking Technical Debt in Agile Low-Code Developments," in *Proc. International Conference on the Quality of Information and Communications Technology (QUATIC)*, Springer, 2021.

[12]. H. M. Sneed, "Dealing with Technical Debt in Agile Development Projects," in *Lecture Notes in Business Information Processing*, vol. 166, B. Franch and P. Soffer, Eds. Berlin, Germany: Springer, 2014, pp. 48–62.

[13]. 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, Mar. 2015.

[14]. A. Martini, V. Stray, and N. B. Moe, "Technical-, Social- and Process Debt in Large-Scale Agile: An Exploratory Case-Study," in *Agile Processes in Software Engineering and Extreme Programming — Workshops (XP 2019)*, Lecture Notes in Business Information Processing, vol. 364, Montreal, Canada, May 2019, pp. 112–119.

[15]. A. Tsintzira, A. Ampatzoglou, A. Chatzigeorgiou, and A. Bibi, "Applying Machine Learning in Technical Debt Management: A Systematic Literature Review," in *Proc. International Conference on the Quality of Information and Communications Technology (QUATIC 2020)*, Faro, Portugal, Sep. 2020, Springer, pp. 97–113.

[16]. M. Mathioudaki, D. Tsoukalas, M. Siavvas, and D. Kehagias, "Technical Debt Forecasting Based on Deep Learning Techniques," in *Proc. 21st International Conference on Computational Science and Its Applications (ICCSA 2021)*, Cagliari, Italy, Sep. 2021, Springer, pp. 96–111.

[17]. L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001.

[18]. D. W. Hosmer and S. Lemeshow, *Applied Logistic Regression*, 2nd ed. New York, NY, USA: Wiley, 2000.

[19]. M. Fowler, *Refactoring: Improving the Design of Existing Code*. Reading, MA, USA: Addison-Wesley, 1999.