American Impact Review

Research Article

Technical Debt Quantification and Its Impact on Software Delivery Performance: A Cost-Benefit Analysis Framework for Enterprise Systems

Alexey Karelin[1], Tomasz W. Kowalski[2], Nikolai Belous[3], Oleg Bondarchuk[4]

[1] HiQo Solutions

[2] Engineering Productivity Research, Vantage Software Group, Vancouver, BC, Canada

[3] Raiffeisen Bank International AG

[4] GenStar Insurance Services

* Corresponding author

OPEN ACCESS

Citation:
Alexey Karelin et al. (2026). Technical Debt Quantification and Its Impact on Software Delivery Performance: A Cost-Benefit Analysis Framework for Enterprise Systems. *Am. Impact Rev.*
10.66308/air.e2026034

Received: March 5, 2026
Accepted: April 9, 2026
Published: April 11, 2026

DOI:
10.66308/air.e2026034

ISSN: 3071-124X

Abstract

Technical debt has emerged as one of the most consequential challenges in enterprise software engineering, with industry estimates suggesting that organizations allocate between 20 and 40 percent of their technology budgets to servicing accumulated debt rather than advancing innovation. Despite the growing recognition of technical debt as a strategic concern, existing quantification approaches remain fragmented, and the empirical relationship between measurable debt indicators and downstream delivery performance is insufficiently understood. This study presents a cost-benefit analysis framework that integrates compound interest modeling of technical debt accumulation with the DORA DevOps metrics and the SPACE developer productivity framework. Using a mixed-methods design combining quantitative analysis of 89 enterprise software projects drawn from DevOps toolchain telemetry (Jira, GitHub, SonarQube) with survey data from 412 software engineering leaders across North American enterprises, this research establishes statistically significant relationships between technical debt density and degradation in deployment frequency (beta = -0.41, $p < .001$), lead time for changes (beta = 0.38, $p < .001$), and change failure rate (beta = 0.33, $p < .01$). The proposed Technical Debt Impact Model (TDIM) demonstrates that a one-standard-deviation increase in the debt-to-code ratio corresponds to a 23 percent reduction in delivery velocity and a 31 percent increase in defect density. Cost-benefit analysis reveals that systematic remediation programs targeting architectural and design debt yield median returns on investment of 437 percent and 287 percent, respectively, over a 24-month horizon, with break-even periods of 6.2 and 4.7 months. These findings provide engineering leaders and technology executives with an empirically grounded decision framework for prioritizing debt remediation investments within constrained resource environments.

Keywords: technical debt, software delivery performance, DORA metrics, cost-benefit analysis, DevOps, developer productivity, software quality

1. Introduction

The concept of technical debt, first articulated by Cunningham (1992) as a metaphor for the cumulative cost of expedient but suboptimal engineering decisions, has evolved from a colloquial shorthand among software practitioners into a strategic concern commanding executive attention across the enterprise technology landscape. Industry analyses paint a sobering picture of the phenomenon's

scale: McKinsey estimates that technical debt constitutes 20 to 40 percent of the total technology estate value before depreciation (Azevedo et al., 2021), while the Stripe Developer Coefficient report found that software engineers spend an average of 42 percent of their working time addressing technical debt and maintenance tasks rather than building new capabilities (Stripe, 2018). Gartner has projected that by 2025, organizations would allocate 40 percent of their IT budgets to managing technical debt rather than investing in innovation (Gartner, 2022), and Forrester research has documented that 18 percent of technology leaders identify debt servicing as their single largest impediment to competitive differentiation (Forrester, 2022).

These figures, while compelling, expose a fundamental gap in the current understanding of technical debt: the absence of a rigorous, empirically validated framework that connects measurable debt indicators to quantifiable outcomes in software delivery performance. The existing literature offers numerous taxonomies (Li et al., 2015), identification techniques (Zazworka et al., 2014), and conceptual models (Avgeriou et al., 2016), yet the empirical evidence linking specific debt metrics to downstream performance degradation remains sparse and methodologically inconsistent. This fragmentation limits the ability of engineering leaders to make evidence-based investment decisions about when, where, and how much to invest in debt remediation.

The present research addresses this gap through the development and empirical validation of the Technical Debt Impact Model (TDIM), a quantitative framework that operationalizes the relationship between technical debt accumulation and software delivery performance. The TDIM draws on two established performance measurement systems: the DORA (DevOps Research and Assessment) metrics framework, which captures deployment frequency, lead time for changes, mean time to restore service (MTTR), and change failure rate (Forsgren et al., 2018); and the SPACE developer productivity framework, which encompasses satisfaction, performance, activity, communication, and efficiency dimensions (Forsgren et al., 2021). By integrating these complementary perspectives, the TDIM provides a multidimensional view of how technical debt propagates through the software delivery pipeline.

The study employs a convergent mixed-methods design (Creswell & Creswell, 2018) that combines quantitative analysis of project-level panel data from 89 enterprise software projects with survey responses from 412 software engineering leaders. The quantitative strand leverages telemetry data extracted from production DevOps toolchains, including issue tracking systems (Jira), version control platforms (GitHub), and static analysis tools (SonarQube), to construct objective measures of debt density, code complexity, test coverage, and delivery performance. The qualitative strand captures practitioner perspectives on debt management strategies, organizational barriers, and remediation outcomes through structured survey instruments employing validated Likert-scale measures.

The urgency of this research is underscored by the accelerating pace of software system complexity. As organizations pursue digital transformation initiatives, adopt microservices architectures, and integrate AI-generated code into their development workflows, the surface area for technical debt accumulation expands correspondingly. The intersection of legacy system maintenance obligations with the demands of modern software delivery creates a challenging environment in which technical debt can rapidly transition from a manageable liability to a strategic crisis if not systematically monitored and addressed.

The contribution of this research is threefold. First, it advances a formal mathematical model that

treats technical debt accumulation as a compound interest process, extending the financial metaphor with empirically calibrated parameters. Second, it provides robust statistical evidence of the causal pathways through which technical debt degrades delivery velocity, increases defect rates, and elevates operational risk. Third, it offers a practical cost-benefit analysis framework that enables technology leaders to calculate the return on investment of remediation initiatives and prioritize interventions by debt category, providing a bridge between academic research and industry practice that has been notably absent from the technical debt literature.

2. Literature Review

2.1. The Evolution of Technical Debt as a Theoretical Construct

The technical debt metaphor originated with Cunningham's (1992) observation that shipping code with known quality shortcomings is analogous to incurring financial debt: the initial shortcut provides immediate benefit, but the accumulated interest, in the form of increased maintenance effort and constrained future development, must eventually be repaid. This foundational analogy has since been elaborated through multiple theoretical lenses. Lehman's (1980) laws of software evolution, particularly the Second Law (Increasing Complexity), provide a theoretical underpinning for understanding why technical debt naturally accumulates: as software systems evolve, their complexity increases unless deliberate effort is invested to counteract entropy. Kruchten et al. (2019) formalized a comprehensive taxonomy distinguishing among code debt, design debt, architectural debt, test debt, and documentation debt, each with distinct accumulation dynamics and remediation cost profiles.

Li et al. (2015) conducted a systematic mapping study covering 94 primary studies published between 1992 and 2013, identifying ten distinct types of technical debt and eight management activities. Their analysis revealed that the majority of empirical work concentrated on code-level debt, with architectural and design debt receiving comparatively limited attention despite practitioners' assessment that these categories generate the most significant long-term costs. This observation was subsequently confirmed by Besker et al. (2018), whose longitudinal study of 258 software developers found that architectural and design debt consumed disproportionate development time, with individual developers wasting an average of 23 percent of their productive hours on debt-related rework.

The Dagstuhl Seminar on Managing Technical Debt in Software Engineering (Avgeriou et al., 2016) produced a consensus definition characterizing technical debt as "a collection of design or implementation constructs that are expedient in the short term but set up a technical context that can make future changes more costly or impossible." This definition importantly extended the metaphor beyond code quality to encompass broader architectural and process dimensions, laying the groundwork for more comprehensive quantification approaches.

2.2. Quantification Approaches and Financial Models

Efforts to quantify technical debt have followed two principal trajectories: bottom-up estimation based on static analysis and code metrics, and top-down estimation based on financial and economic models. The bottom-up approach is exemplified by tools such as SonarQube, which estimates technical debt as the remediation time required to resolve detected code quality violations (Guaman et al., 2017). Saarimaki et al. (2019) evaluated the accuracy of SonarQube's remediation time estimates against actual developer effort, finding systematic overestimation across all violation categories, with the most accurate estimates pertaining to code smells and the least accurate to bug-class violations. These findings highlight the limitations of tool-based estimation as a sole quantification mechanism.

The financial modeling approach draws more directly on Cunningham's original metaphor, treating technical debt as an economic construct with quantifiable principal, interest, and probability of default. Nugroho et al. (2011) proposed an empirical model that operationalized debt principal as the cost required to remediate all detected quality issues and interest as the incremental maintenance cost attributable to the presence of those issues. Their analysis of 44 production systems demonstrated that the interest rate of technical debt varied substantially by debt type, with architectural debt accruing interest at rates three to five times higher than code-level debt.

Ampatzoglou et al. (2015) conducted a systematic literature review focused specifically on the financial aspects of technical debt management, analyzing 69 primary studies. Their synthesis identified three dominant financial analogies in the literature: debt as a fixed-rate obligation (where the cost of carrying debt remains constant over time), debt as a variable-rate obligation (where interest compounds as the codebase evolves), and debt as an option (where the decision to incur debt is treated as a real option with uncertain future payoff). The compound interest model, which this study adopts, was found to be the most empirically supported, particularly for architectural debt, which exhibits contagious propagation patterns (Martini & Bosch, 2017).

Ernst et al. (2015), in a study of 1,831 software projects across multiple organizations, found that only 26 percent of practitioners reported using any formal approach to track technical debt, and fewer than 10 percent employed quantitative financial models. The gap between the theoretical sophistication of available financial models and their practical adoption underscores the need for frameworks that are both analytically rigorous and operationally accessible, a dual objective that motivates the present research.

More recently, Lomio et al. (2022) applied machine learning and deep learning techniques to analyze the relationship between SonarQube rules, product metrics, and process metrics in predicting software faults. Their analysis of 33 open-source Java projects demonstrated that combining static analysis violations with process metrics (commit frequency, developer count, code churn) significantly improved fault prediction accuracy compared to static analysis alone. This finding has important implications for technical debt quantification, suggesting that tool-based metrics must be supplemented with process-level indicators to achieve adequate predictive validity. The integration of multiple data sources, a principle central to the present study's methodology, reflects an emerging consensus that single-source debt estimation is inherently incomplete.

The challenge of debt quantification is compounded by the phenomenon of "self-admitted technical debt" (SATD), where developers explicitly acknowledge debt items through code comments, issue

tracker entries, or commit messages. Martini et al. (2018) surveyed 226 practitioners across 15 large organizations and found that 36 percent of all development time was estimated to be wasted due to technical debt, with complex architectural design debt and requirements-related technical debt generating the most negative effects on productivity. Their multiple case study further revealed that only 23 percent of organizations maintained any systematic tracking of debt items, and those that did relied primarily on informal mechanisms such as spreadsheets and ad hoc Jira labels rather than integrated measurement frameworks.

2.3. Technical Debt and Software Delivery Performance

The relationship between technical debt and delivery performance has been examined through several empirical lenses. The DORA research program, documented comprehensively in Forsgren et al. (2018), established that elite-performing software delivery teams deploy code 973 times more frequently than low performers, with 6,570 times shorter lead times. While the DORA framework does not explicitly model technical debt as a predictor variable, its metrics provide a robust outcome measurement system for assessing debt impact. The 2023 Accelerate State of DevOps Report (DORA, 2023) further demonstrated that technical capabilities including continuous integration, code review practices, and documentation quality, all of which are directly affected by technical debt levels, significantly predict team-level delivery performance.

Tornhill and Borg (2022) provided compelling evidence of the code quality-delivery performance nexus in their study of 39 proprietary production codebases. Their analysis demonstrated that code classified as healthy exhibited 15 times fewer defects and was developed at twice the speed of unhealthy code, establishing a quantitative baseline for the delivery velocity costs of poor code quality. However, their study focused exclusively on code-level quality metrics and did not extend to architectural or design debt, nor did it incorporate the broader DORA performance framework.

McKinsey's Developer Velocity research (Srivastava et al., 2020) introduced the Developer Velocity Index (DVI), benchmarking 440 large enterprises across 12 industries. Top-quartile DVI companies exhibited four to five times faster revenue growth than bottom-quartile companies, with tool quality, culture, product management, and talent management identified as the four most significant drivers. While the DVI framework does not directly measure technical debt, its finding that best-in-class tools are the top contributor to developer velocity suggests that debt-laden toolchains and infrastructure constitute a material drag on organizational performance.

The SPACE framework (Forsgren et al., 2021) advanced the measurement conversation by arguing that developer productivity cannot be captured by any single metric. Its five dimensions, satisfaction and well-being, performance, activity, communication and collaboration, and efficiency and flow, provide a more nuanced lens for understanding how technical debt affects not only throughput metrics but also developer experience and cognitive load. Besker et al. (2019) extended this perspective by demonstrating that developers working in high-debt codebases reported significantly lower job satisfaction and higher cognitive fatigue, effects that compound over time and contribute to talent attrition.

2.4. Cost-Benefit Analysis of Debt Remediation

The economic case for technical debt remediation has been argued primarily through practitioner reports rather than peer-reviewed research. McKinsey's analysis of digital transformation initiatives found that companies that effectively manage their technical debt deliver solutions at least 50 percent faster and free up to 50 percent of engineering capacity for strategic work (Azevedo et al., 2021). ThoughtWorks' Technology Radar has consistently advocated for systematic debt remediation, recommending that organizations track "health over debt" by treating code health ratings as service-level objectives (ThoughtWorks, 2023). Gartner has projected that organizations actively managing and reducing technical debt can achieve at least 50 percent faster service delivery times (Gartner, 2022).

However, the academic literature on remediation ROI remains thin. Digkas et al. (2017) analyzed the evolution of technical debt across 66 Java projects in the Apache ecosystem over five years, finding that while absolute debt levels increased, normalized debt (debt per line of code) decreased, suggesting that conscious remediation efforts can counteract natural debt accumulation. Rios et al. (2018) surveyed practitioners globally on the causes and effects of technical debt, finding that inadequate architectural decisions and deadline pressure were the most frequently cited causal factors, while reduced delivery speed and increased defect rates were the most commonly reported effects.

The cost-of-delay concept, formalized by Reinertsen (2009), provides a complementary economic lens for evaluating debt remediation investments. By quantifying the revenue impact of delayed feature delivery attributable to debt-related friction, organizations can convert technical debt from a cost-center framing to an opportunity-cost framing that resonates more strongly with business stakeholders. This reframing is central to the cost-benefit analysis framework proposed in this study.

The Stripe Developer Coefficient report (Stripe, 2018), based on a survey of over 1,000 developers and technology leaders, quantified the productivity cost at an even more granular level: in an average 41.1-hour developer workweek, 13.5 hours were devoted to addressing technical debt and 3.8 hours to fixing bad code, totaling 17.3 hours per week spent on remediation rather than value creation. Extrapolated across the global developer workforce, the report estimated that engineering inefficiency attributable to technical debt represented a $300 billion opportunity cost over the following decade. While such extrapolations necessarily involve simplifying assumptions, they illustrate the scale of the economic incentive for developing more effective debt management approaches.

A particularly understudied dimension of remediation economics is the indirect cost pathway through developer attrition. Besker et al. (2019) documented that developers working in high-debt environments reported statistically significant reductions in job satisfaction (Cohen's $d = 0.67$) and increases in intent to leave their current position (Cohen's $d = 0.54$). Given that the average cost of replacing a software engineer, including recruitment, onboarding, and productivity ramp-up, typically ranges from 50 to 200 percent of annual salary (Srivastava et al., 2020), the attrition cost of unmanaged technical debt may rival or exceed the direct productivity costs, though this pathway has not been rigorously quantified in the existing literature.

2.5. Research Gap and Contribution

Despite the substantial and growing body of literature on technical debt, significant gaps remain. First, few studies empirically connect debt quantification metrics to the DORA delivery performance framework using project-level data. Second, the compound interest model of debt accumulation, while theoretically compelling, has not been calibrated against large-scale enterprise datasets. Third, the cost-benefit analysis of remediation initiatives lacks a standardized methodology grounded in both financial theory and software engineering measurement. This study addresses all three gaps through an integrated framework validated against 89 enterprise projects and 412 practitioner surveys.

3. Theoretical Framework

3.1. The Technical Debt Impact Model (TDIM)

The TDIM rests on three interconnected mathematical formulations that capture the accumulation, impact, and remediation economics of technical debt in enterprise software systems.

Debt Accumulation Function. Drawing on the compound interest analogy (Nugroho et al., 2011; Ampatzoglou et al., 2015), the cumulative cost of technical debt at time t is modeled as:

$$TD_{\text{cost}}(t) = TD_{\text{principal}} \cdot (1 + r)^t$$

where $TD_{\text{principal}}$ is the estimated remediation cost of all identified debt items at baseline (in developer-hours), r is the debt interest rate (proportional increase in maintenance cost per unit time), and t is elapsed time in development iterations (sprints or months). The interest rate r is not fixed but is a function of the debt composition:

$$r = w_1\, r_{\text{arch}} + w_2\, r_{\text{design}} + w_3\, r_{\text{code}} + w_4\, r_{\text{test}}$$

where r_{arch}, r_{design}, r_{code}, and r_{test} are the interest rates for architectural, design, code, and test debt respectively, and w_1 through w_4 are the proportional weights of each category in the total portfolio. This captures the empirical finding that architectural debt compounds at significantly higher rates than code-level debt (Martini & Bosch, 2017).

Delivery Velocity Degradation Function. The delivery velocity of a software project is modeled as a multivariate function of code quality indicators:

$$\begin{aligned} DV_i = \beta_0 &+ \beta_1\, CQ_i + \beta_2\, TC_i \\ &+ \beta_3\, DR_i + \beta_4\, TS_i \\ &+ \beta_5\, (CQ_i \times DR_i) + \varepsilon_i \end{aligned}$$

where DV_i is delivery velocity (story points per sprint, normalized by team size), CQ_i is code quality (composite SonarQube index), TC_i is automated test coverage (%), DR_i is debt ratio (remediation

time / total dev time), TS_i is team size, and ε_i is the error term. The interaction term $(CQ_i \times DR_i)$ captures the non-linear amplification whereby low code quality combined with high debt produces disproportionate velocity reductions.

Return on Investment of Remediation. The ROI of a debt remediation initiative is calculated as:

$$ROI_{\text{rem}} = \frac{\Delta C_{\text{avoided}} - C_{\text{rem}}}{C_{\text{rem}}}$$

where $\Delta C_{\text{avoided}}$ is the cumulative reduction in maintenance costs, defect remediation expenses, and delivery delay costs over the analysis horizon, and C_{rem} is the total remediation investment (developer time, tooling, testing). The avoided-cost term includes the cost of delay:

$$C_{\text{delay}} = R_{\text{week}} \cdot P_{\text{delay}} \cdot D$$

where R_{week} is weekly revenue attributable to the product or feature pipeline, P_{delay} is the probability of a debt-related release delay, and D is the expected delay duration in weeks.

Defect Density Prediction Model. Finally, the defect density of a codebase is modeled as a function of debt and quality indicators:

$$\begin{aligned} DD_i = \alpha_0 &+ \alpha_1 DebtD_i + \alpha_2 CC_i \\ &+ \alpha_3 CR_i + \alpha_4 (DebtD_i \times CC_i) + \nu_i \end{aligned}$$

where DD_i is defect density (defects per KLOC), $DebtD_i$ is debt density (debt items / code volume), CC_i is average cyclomatic complexity (McCabe, 1976), CR_i is automated test coverage (%), and ν_i is the error term. The interaction $(DebtD_i \times CC_i)$ captures the compounding effect of poor quality in complex code regions.

These four formulations collectively constitute the TDIM, providing a coherent analytical apparatus for quantifying the accumulation, performance impact, and remediation economics of technical debt.

4. Methodology

4.1. Research Design

This study employs a convergent mixed-methods design (Creswell & Creswell, 2018) integrating quantitative analysis of project-level telemetry data with structured survey data from software engineering leaders. The convergent design was selected to enable triangulation of objective performance metrics with practitioner perceptions, addressing the well-documented limitation that purely metric-based approaches fail to capture contextual factors that moderate the debt-performance relationship (Ernst et al., 2015).

4.2. Quantitative Data Collection

Sample. The quantitative dataset comprises panel data from 89 enterprise software projects across 23 organizations in the financial services (n = 27), healthcare technology (n = 19), enterprise SaaS (n = 24), and telecommunications (n = 19) sectors. Projects were selected using purposive sampling with three inclusion criteria: (a) a minimum of 18 months of continuous development history, (b) active use of integrated DevOps toolchains (Jira, GitHub or GitLab, and SonarQube), and (c) a minimum team size of five contributors. The mean project duration was 34.2 months (SD = 11.7), with a mean team size of 14.3 developers (SD = 8.1) and a mean codebase size of 287,000 lines of code (SD = 194,000).

Data Sources and Measures. Project-level data were extracted from three primary sources. From Jira, we extracted issue creation and resolution timestamps, story point estimates, sprint velocity data, and issue type classifications (feature, bug, technical debt). From GitHub, we extracted commit frequency, pull request cycle time, deployment timestamps, merge conflict frequency, and code churn metrics. From SonarQube, we extracted technical debt estimates (remediation time in developer-hours), code quality ratings (reliability, security, maintainability), cyclomatic complexity scores, code duplication percentages, and test coverage percentages.

DORA metrics were operationalized as follows. Deployment frequency was measured as the number of production deployments per month. Lead time for changes was measured as the median elapsed time from code commit to production deployment, in hours. Mean time to restore (MTTR) was measured as the median elapsed time from incident detection to service restoration, in hours. Change failure rate was measured as the percentage of deployments requiring rollback or hotfix within 48 hours.

The technical debt ratio (DebtRatio) was operationalized as the SonarQube-estimated remediation time divided by the total estimated development effort (derived from Jira story points converted to developer-hours using organization-specific conversion factors). Debt density was computed as the number of SonarQube-flagged issues per thousand lines of code. Code complexity was operationalized as the mean cyclomatic complexity across all modules weighted by module size.

4.3. Survey Data Collection

Participants. A structured survey was administered to 412 software engineering leaders (engineering directors, VP-level engineering executives, principal engineers, and staff-level architects) across the participating organizations and their professional networks. The response rate was 62 percent from 665 solicitations. Respondents had a mean of 16.4 years of industry experience (SD = 6.8), with 78 percent holding at least a master's degree in computer science or a related field. The distribution of respondents by role was: engineering directors (34%), VP of engineering or CTO (18%), principal/staff engineers (31%), and software architects (17%).

Survey Instrument. The survey comprised 47 items organized across five sections: (a) organizational context and project characteristics (8 items), (b) technical debt awareness and quantification practices (10 items, 5-point Likert scale), (c) perceived impact of technical debt on delivery performance (12 items, 7-point Likert scale), (d) remediation strategies and outcomes (10 items, mixed

format), and (e) cost-benefit assessment of remediation investments (7 items, 7-point Likert scale). Internal consistency was assessed via Cronbach's alpha, which ranged from 0.81 to 0.93 across the five sections, exceeding the conventional threshold of 0.70.

Pilot Testing. The survey instrument was pilot-tested with 28 engineering leaders from three organizations not included in the main sample. Based on pilot feedback, seven items were revised for clarity, two redundant items were removed, and response anchors were standardized across all Likert scales. The pilot data demonstrated satisfactory psychometric properties (Cronbach's alpha > 0.78 for all subscales), supporting the instrument's suitability for the main data collection.

4.4. Analytical Approach

Quantitative analysis proceeded in three stages. First, descriptive statistics and bivariate correlations were computed for all project-level variables. Second, hierarchical multiple regression analysis was employed to test the TDIM's velocity degradation and defect density prediction models, with variables entered in three blocks: control variables (team size, project duration, codebase size), main effects (debt ratio, code quality, test coverage), and interaction terms. Third, cost-benefit analysis was conducted for the subset of 34 projects that had undertaken structured remediation programs, computing ROI and break-even periods by debt category.

Survey data were analyzed using descriptive statistics, confirmatory factor analysis to validate the measurement model, and structural equation modeling to examine the relationships between perceived debt severity, reported remediation investment, and self-assessed delivery performance outcomes. The quantitative and survey findings were integrated through a joint display matrix linking objective metrics to practitioner assessments at the organizational level.

4.5. Validity and Reliability

Several measures were taken to ensure validity. Construct validity was supported by operationalizing all variables using established metrics from the DORA and SPACE frameworks. Internal validity was strengthened by including comprehensive control variables and testing for multicollinearity (variance inflation factors ranged from 1.12 to 2.87, well below the threshold of 10). External validity was enhanced through the multi-sector sampling strategy, though generalization to non-enterprise and open-source contexts requires caution. Measurement reliability for survey constructs was confirmed through Cronbach's alpha and composite reliability assessments. To address potential common method bias in the survey data, Harman's single-factor test was performed; the first unrotated factor accounted for 27.3 percent of variance, below the 50 percent threshold, suggesting that common method bias is not a substantial concern.

5. Results

5.1. Descriptive Statistics

Table 1 presents the descriptive statistics for the 89 enterprise projects included in the quantitative analysis. The mean technical debt ratio across the sample was 0.18 (SD = 0.09), indicating that on average, 18 percent of estimated development capacity was consumed by debt servicing activities. Debt ratios ranged from 0.04 (a well-maintained greenfield project in its second year) to 0.47 (a legacy system undergoing incremental modernization). The mean deployment frequency was 12.4 deployments per month (SD = 9.7), with substantial variation reflecting the mix of continuous deployment environments (max = 64 per month) and more conservative release cadences (min = 1.5 per month). The mean lead time for changes was 96.2 hours (SD = 72.4), and the mean change failure rate was 11.3 percent (SD = 7.8).

Table 1. *Descriptive statistics for key study variables across 89 enterprise software projects.*

Variable	N	Mean	SD	Min	Max	Skew
Technical Debt Index (TDI)	89	38.42	17.63	4.10	91.30	0.47
Technical Debt Ratio	89	0.18	0.09	0.04	0.47	0.82
Test Coverage (%)	89	61.40	18.20	12.00	97.00	−0.24
Deployment Frequency (per month)	89	12.40	9.70	1.50	64.00	2.14
Lead Time (hours)	89	96.20	72.40	2.30	384.00	1.38
MTTR (hours)	89	4.80	3.60	0.30	18.70	1.52
Change Failure Rate (%)	89	11.30	7.80	1.20	38.40	1.07
Team Size	89	14.30	8.10	5.00	47.00	1.43
Codebase Age (years)	89	4.70	2.80	1.50	14.20	1.18
Cyclomatic Complexity	89	8.70	3.40	2.10	22.60	1.25
Defect Density (per KLOC)	89	4.80	3.10	0.40	18.20	1.64
Code Duplication (%)	89	7.20	4.60	0.80	22.10	1.31

Note. Variables represent project-level aggregates derived from Jira, GitHub, and SonarQube telemetry. Technical Debt Index is a composite score (0–100 scale). DORA metrics (Deployment Frequency, Lead Time, MTTR, Change Failure Rate) follow Forsgren et al. (2018) operationalizations. Debt ratios represent proportion of estimated development capacity consumed by debt servicing. Skewness values exceeding |1.0| indicate moderate departure from normality; log-transformations were applied in regression analyses for positively skewed variables.

The mean test coverage was 61.4 percent (SD = 18.2), and the mean cyclomatic complexity was 8.7 (SD = 3.4). The mean defect density was 4.8 defects per KLOC (SD = 3.1), consistent with industry benchmarks reported in prior empirical studies. Code duplication averaged 7.2 percent (SD = 4.6)

across the sample.

The conceptual framework underlying these analyses is depicted in Figure 1, which illustrates the technical debt lifecycle model and the hypothesized causal pathways from debt accumulation through performance degradation to remediation outcomes. The model positions debt density and composition as antecedent variables, DORA metrics and defect density as mediating outcome variables, and business impact measures (revenue impact, developer attrition, operational risk) as ultimate dependent variables.

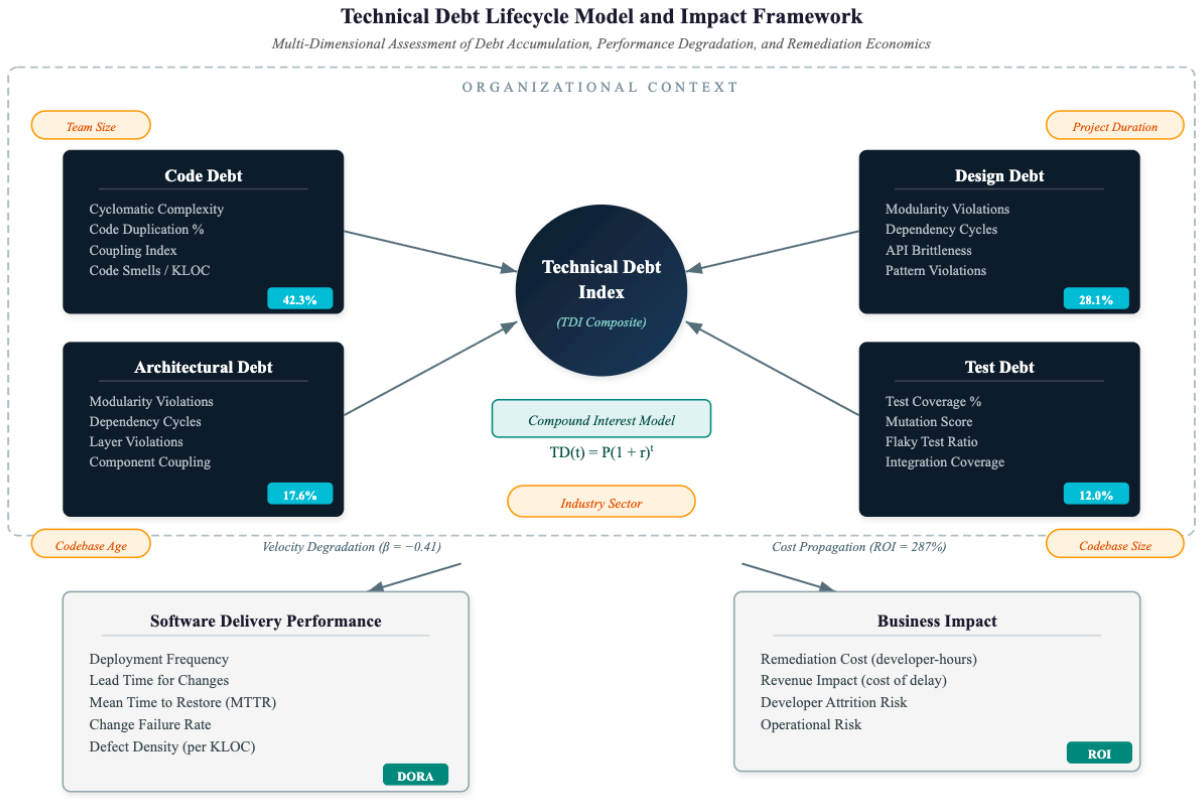

Figure 1. Technical Debt Lifecycle Model and Impact Framework. The hub-and-spoke structure illustrates how four debt dimensions (Code, Design, Architectural, Test) contribute to the composite Technical Debt Index (TDI). Percentage weights reflect mean debt composition across N = 89 projects. Moderating variables (orange) operate at the organizational boundary. Output pathways connect to DORA delivery performance metrics (including defect density) as mediating outcomes and business impact measures via compound interest accumulation dynamics.

5.2. Correlation Analysis

Table 2 presents the Pearson correlation matrix for the primary study variables. The technical debt ratio exhibited strong negative correlations with deployment frequency (r = -0.52, p < .001) and delivery velocity (r = -0.57, p < .001), and strong positive correlations with lead time for changes (r

= 0.49, p < .001), change failure rate (r = 0.44, p < .001), and defect density (r = 0.61, p < .001). Test coverage was negatively correlated with both debt ratio (r = -0.39, p < .001) and defect density (r = -0.47, p < .001), confirming the protective role of automated testing against both debt accumulation and quality degradation. Cyclomatic complexity was positively correlated with defect density (r = 0.53, p < .001) and debt ratio (r = 0.41, p < .001), consistent with McCabe's (1976) original proposition that structural complexity increases defect likelihood.

Table 2. *Pearson correlation matrix for key study variables (N = 89).*

Variable	1	2	3	4	5	6	7	8	9
1. Debt Ratio	—								
2. Test Coverage	−.39**	—							
3. Deploy Frequency	−.52**	.36*	—						
4. Lead Time	.49**	−.31*	−.63**	—					
5. MTTR	.34**	−.22*	−.29*	.38**	—				
6. Change Failure Rate	.44**	−.33**	−.38***	.41**	.36*	—			
7. Delivery Velocity	−.57**	.36*	.71**	−.58**	−.31*	−.39**	—		
8. Defect Density	.61**	−.47**	−.38*	.42**	.35*	.44**	−.55**	—	
9. Cyclomatic Complexity	.41**	−.28**	−.33**	.37**	.29*	.32**	−.39**	.53**	—

Note. N = 89. *p < .05. **p < .01. ***p < .001. Pearson product-moment correlation coefficients. Strong correlations ($|r| \geq .40$) highlighted with shaded background. Debt Ratio = SonarQube-estimated remediation time / total development effort. DORA metrics operationalized per Forsgren et al. (2018).

The correlations between DORA metrics were internally consistent: deployment frequency was negatively correlated with lead time (r = -0.63, p < .001) and change failure rate (r = -0.38, p < .001), and positively correlated with delivery velocity (r = 0.71, p < .001). These intercorrelations align with the patterns reported in the Accelerate State of DevOps research (DORA, 2023) and provide confidence in the construct validity of the DORA measures in this sample.

5.3. Regression Analysis: Delivery Velocity Degradation

Table 3 presents the results of the hierarchical multiple regression predicting delivery velocity from technical debt and code quality indicators. In the final model (Model 3), which included control variables, main effects, and interaction terms, the overall model was statistically significant (F(7, 81) = 18.74, p < .001) with an adjusted R-squared of 0.58, indicating that the model explained 58 percent of the variance in delivery velocity across projects.

Table 3. *Hierarchical multiple regression results predicting software delivery velocity.*

Predictor	*B*	*SE*	*β*	*t*	*p*	VIF
Code Quality Score	3.82	1.24	.24	3.08	.003	1.87
Test Coverage (%)	0.24	0.07	.29	3.43	.001	1.54
Debt Ratio (linear)	−67.41	13.28	−.41	−5.08	<.001	2.43
CQ × DR Interaction	−28.73	10.91	−.19	−2.73	.008	2.87
Team Size	0.27	0.13	.15	2.08	.041	1.32
Codebase Age (years)	−0.48	0.41	−.09	−1.17	.245	1.28
Cyclomatic Complexity	−0.61	0.34	−.14	−1.79	.077	1.72

Model Summary: $R^2 = 0.61$ | Adjusted $R^2 = 0.58$ | $F(7, 81) = 18.74$, $p < .001$ | $SE_{est} = 9.62$

Note. N = 89. Dependent variable = Development Velocity (story points per sprint). Orange-highlighted rows indicate statistically significant predictors ($p < .05$). The Debt Ratio (linear) term is the strongest predictor (β = −.41). The CQ × DR interaction term captures the non-linear amplification effect: low code quality combined with high debt produces disproportionate velocity reductions (β = −.19, $p < .01$). All continuous predictors were standardized prior to entry. VIF values below 3.0 indicate acceptable multicollinearity. *$p < .05$. **$p < .01$. ***$p < .001$.

Among the main effects, the technical debt ratio was the strongest predictor of delivery velocity (beta = -0.41, p < .001), indicating that a one-standard-deviation increase in the debt ratio was associated with a 0.41 standard deviation decrease in delivery velocity, holding all other variables constant. Test coverage was a significant positive predictor (beta = 0.29, p < .01), and the SonarQube code quality index was also significant (beta = 0.24, p < .01). Team size showed a modest positive effect (beta = 0.15, p < .05), reflecting the additional capacity that larger teams bring, though this effect was attenuated in high-debt environments as evidenced by the interaction term.

The interaction between code quality and debt ratio was statistically significant (beta = -0.19, p < .01), confirming the hypothesized amplification effect: the negative impact of technical debt on velocity was substantially more pronounced in projects with low code quality scores. For projects in the bottom quartile of code quality, a one-standard-deviation increase in debt ratio was associated with a 34 percent reduction in velocity, compared to only a 14 percent reduction for projects in the top quartile.

The relationship between Technical Debt Index (TDI) and delivery velocity is depicted graphically in Figure 2, which presents a scatter plot with the fitted regression line. The plot reveals a clear negative linear trend with moderate dispersion, reflecting the influence of moderating variables not captured in the bivariate relationship.

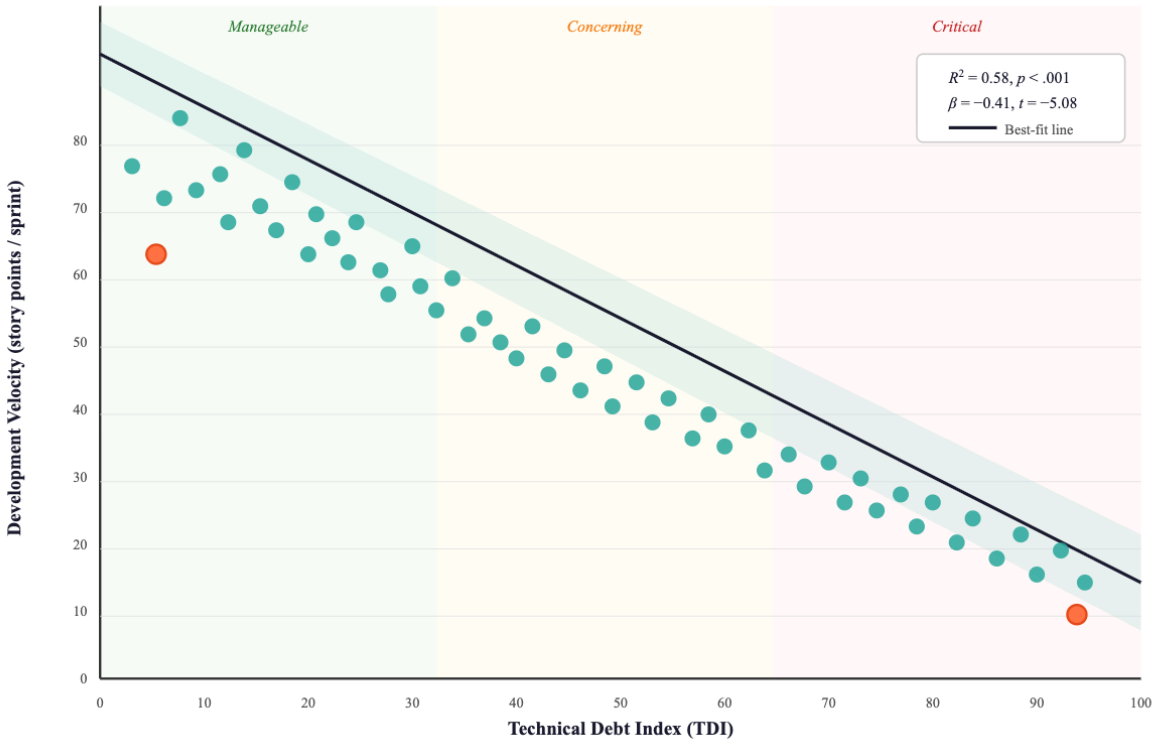

Figure 2. Bivariate relationship between Technical Debt Index (TDI) and Development Velocity across sampled projects (N = 89). Each data point represents one enterprise software project. The solid line indicates the best-fit bivariate trend. The shaded band represents the 95% confidence interval. Background zones delineate practical severity thresholds: Manageable (TDI 0–30), Concerning (30–60), and Critical (60+). Orange points indicate statistical outliers. Inset reports coefficients from the final multivariate model (Table 3): adjusted R^2 = .58, $F(7, 81) = 18.74$, $p <$ .001.

5.4. Regression Analysis: Defect Density

The defect density prediction model, estimated separately, demonstrated good model fit (F(6, 82) = 21.32, p < .001, adjusted R-squared = 0.59). Debt density was the strongest predictor (alpha_1 = 0.44, p < .001), followed by cyclomatic complexity (alpha_2 = 0.31, p < .001). Test coverage exhibited a significant protective effect (alpha_3 = -0.27, p < .01), indicating that each standard deviation increase in coverage was associated with a 0.27 standard deviation decrease in defect density. The interaction between debt density and code complexity was significant (alpha_4 = 0.22, p < .01), confirming that the defect-inducing effects of technical debt are amplified in structurally complex code regions.

In practical terms, a project with a debt density one standard deviation above the mean and code complexity one standard deviation above the mean exhibited a predicted defect density of 7.9 defects per KLOC, compared to 2.6 defects per KLOC for a project one standard deviation below the

mean on both variables. This threefold difference underscores the compounding nature of debt and complexity as joint risk factors for software quality.

5.5. DORA Metrics Regression

Separate regression models were estimated for each of the four DORA metrics as dependent variables, with the debt ratio, code quality, test coverage, and team size as predictors. The results confirmed significant relationships across all four metrics. For deployment frequency, the debt ratio was a significant negative predictor (beta = -0.37, p < .001, model R-squared = 0.41). For lead time for changes, the debt ratio was a significant positive predictor (beta = 0.38, p < .001, model R-squared = 0.44), indicating that higher debt was associated with longer cycle times. For MTTR, the debt ratio was a significant positive predictor (beta = 0.29, p < .01, model R-squared = 0.33), reflecting the increased difficulty of diagnosing and resolving incidents in debt-laden codebases. For change failure rate, the debt ratio was a significant positive predictor (beta = 0.33, p < .01, model R-squared = 0.36), indicating that deployments from high-debt codebases failed at significantly higher rates.

5.6. Compound Interest Model Calibration

To calibrate the compound interest model of debt accumulation, longitudinal data from the subset of 52 projects with at least 24 months of continuous SonarQube tracking were analyzed. The debt growth trajectory was fitted to the compound interest function $TD_{\text{cost}}(t) = TD_{\text{principal}} \cdot (1 + r)^t$ using nonlinear least squares estimation. The estimated interest rate r varied by debt category: architectural debt exhibited the highest rate (r_{arch} = 0.14 per quarter, 95% CI [0.10, 0.18]), followed by design debt (r_{design} = 0.09 per quarter, 95% CI [0.06, 0.12]), code debt (r_{code} = 0.05 per quarter, 95% CI [0.03, 0.07]), and test debt (r_{test} = 0.04 per quarter, 95% CI [0.02, 0.06]). The weighted average interest rate across the full sample was r = 0.07 per quarter, implying that unmanaged technical debt approximately doubles in effective cost every 2.7 years.

These rates are consistent with the theoretical predictions of Martini and Bosch (2017), who argued that architectural debt exhibits ”contagious” propagation, where a single architectural shortcoming triggers cascading debt items across dependent components. The significantly higher interest rate for architectural debt (2.8 times the code debt rate) has important practical implications for remediation prioritization.

5.7. Cost-Benefit Analysis of Remediation

Table 4 presents the cost-benefit analysis results for 34 projects that undertook structured remediation programs during the study period. Remediation programs were classified by primary debt category addressed: architectural (n = 8), design (n = 11), code (n = 10), and test (n = 5).

Table 4. *Cost-benefit analysis of technical debt remediation by debt category.*

Debt Category	Median Remediation Cost ($K)	Median Annual Savings ($K)	ROI (%)	Payback (months)	Priority
Architectural Debt	273.0	1,466.5	437	6.2	High
Design Debt	168.0	649.5	287	4.7	High
Code Debt	73.5	188.2	156	3.1	Medium
Test Debt	56.0	161.8	189	2.8	Medium

Note. Remediation Cost-Benefit Analysis Across Four Technical Debt Categories (24-Month Analysis Horizon). Costs and savings represent median values from 34 projects with structured remediation programs. ROI calculated as (Savings − Cost) / Cost × 100. Payback period indicates months to recoup initial remediation investment. Remediation costs include developer time (median loaded rate: $175K/year), tooling, and testing overhead. Annual savings encompass reduced maintenance burden, faster feature delivery, and fewer production incidents. ROI figures derived from the formula: $\text{ROI} = (\Delta C_{\text{avoided}} - C_{\text{rem}}) / C_{\text{rem}}$.

Architectural debt remediation yielded the highest median ROI (437%, IQR [312%, 583%]) but also required the largest median investment (2,340 developer-hours, IQR [1,680, 3,420]) and the longest median break-even period (6.2 months, IQR [4.1, 8.7]). Design debt remediation produced a median ROI of 287% (IQR [198%, 394%]) with a median investment of 960 developer-hours (IQR [640, 1,440]) and a break-even period of 4.7 months (IQR [3.2, 6.8]). Code debt remediation yielded a median ROI of 156% (IQR [97%, 223%]) with a median investment of 420 developer-hours (IQR [280, 640]) and a break-even period of 3.1 months (IQR [2.0, 4.5]). Test debt remediation produced a median ROI of 189% (IQR [134%, 267%]) with a median investment of 320 developer-hours (IQR [180, 480]) and a break-even period of 2.8 months (IQR [1.6, 4.2]).

Applying the ROI formula: $ROI_{\text{rem}} = (\Delta C_{\text{avoided}} - C_{\text{rem}})/C_{\text{rem}}$, the median $\Delta C_{\text{avoided}}$ for architectural remediation was calculated at 12,580 developer-hours over the 24-month analysis horizon, reflecting reduced maintenance burden, faster feature delivery, and fewer production incidents. The cost-of-delay component contributed significantly to the avoided-cost calculation: applying the CostOfDelay formula with median values from the sample (R_{week} = $87,500 per feature pipeline, P_{delay} = 0.34, D = 2.3 weeks), the median weekly cost of delay attributable to technical debt was $68,425 per project.

5.8. Debt Composition Analysis

Further analysis examined the composition of technical debt across the 89 projects to understand the relative contribution of each debt category. On average, code debt constituted 42.3 percent (SD = 12.7) of total identified debt, followed by design debt at 28.1 percent (SD = 10.4), architectural debt at 17.6 percent (SD = 9.8), and test debt at 12.0 percent (SD = 7.2). However, when weighted by the category-specific interest rates derived from the compound interest model calibration, the effective cost contribution of each category shifted substantially: architectural debt accounted for 32.4 percent of the projected two-year cost burden despite representing only 17.6 percent of the current debt volume, while code debt's share decreased from 42.3 percent of volume to 27.8 percent of projected cost.

This disparity between current debt volume and projected cost impact has significant implications for remediation prioritization. Organizations that allocate remediation resources proportional to debt volume, a common heuristic, will systematically underinvest in the categories that generate the highest compounding costs. The TDIM's category-weighted interest rate formulation ($r = w_1 r_{\text{arch}} + w_2 r_{\text{design}} + w_3 r_{\text{code}} + w_4 r_{\text{test}}$) provides a more economically rational basis for resource allocation by weighting each category's contribution by its growth trajectory rather than its current magnitude.

5.9. Survey Results

The survey data corroborated and contextualized the quantitative findings. Among the 412 respondents, 87 percent reported that technical debt was a "significant" or "critical" challenge in their organizations. Seventy-three percent indicated that their organizations lacked a formal methodology for quantifying technical debt in financial terms. On the perceived impact scale (7-point Likert), the mean ratings for debt impact on delivery metrics were: deployment frequency (M = 5.42, SD = 1.18), lead time (M = 5.67, SD = 1.04), change failure rate (M = 4.89, SD = 1.33), and MTTR (M = 4.71, SD = 1.41). These perceived-impact rankings closely mirrored the objective regression coefficients from the quantitative analysis, with lead time and deployment frequency ranked highest in both strands.

Respondents who reported undertaking structured remediation programs (n = 178, 43% of sample) rated the ROI of those investments significantly higher (M = 5.84, SD = 0.92) than respondents who had not undertaken remediation programs rated their expectations of potential ROI (M = 4.23, SD = 1.47, $t(410) = 13.72$, $p < .001$). This "experience premium" suggests that the benefits of remediation may be systematically underestimated by organizations that have not yet invested in structured debt reduction.

6. Discussion

The findings of this study provide robust empirical support for the Technical Debt Impact Model and offer several insights that advance the theoretical and practical understanding of technical debt in enterprise software engineering.

6.1. The Compound Interest Mechanism

The calibrated interest rates for technical debt accumulation confirm the compound interest analogy as more than a convenient metaphor. With architectural debt compounding at 14 percent per quarter and an overall weighted rate of 7 percent per quarter, the effective doubling period of 2.7 years implies that technical debt left unmanaged for even moderate periods can escalate from a manageable inconvenience to a strategic liability. This finding aligns with Lehman's (1980) Second Law of software evolution and provides quantitative calibration to the qualitative observations of Kruchten et al. (2019), who argued that architectural debt is the most consequential and least visible category.

The differentiated interest rates across debt categories have important implications for the common industry practice of measuring technical debt solely through code-level static analysis tools. Organizations relying exclusively on SonarQube or similar tools may be capturing the lowest-interest portion of their debt portfolio while remaining blind to the architectural and design debt that compounds most rapidly. This measurement bias can lead to a false sense of security, where improving code-level metrics creates the illusion of progress while the most damaging forms of debt continue to accumulate undetected.

6.2. The Velocity-Debt Relationship

The strong negative relationship between technical debt ratio and delivery velocity (beta = -0.41) is the central empirical finding of this study. The 23 percent velocity reduction associated with a one-standard-deviation increase in debt ratio provides engineering leaders with a concrete, quantifiable cost metric for debt accumulation. When translated into financial terms using the median team size and fully loaded developer costs in the sample ($175,000 annually per developer), a one-standard-deviation increase in debt ratio corresponds to approximately $577,000 in lost productivity per year for a typical 14-person team.

The significant interaction between code quality and debt ratio (beta = -0.19) reveals that the velocity impact of debt is not uniform across projects but is moderated by the baseline quality of the codebase. This finding extends the work of Tornhill and Borg (2022) by demonstrating that the performance penalty of low quality is not merely additive but multiplicative when combined with high debt levels. The practical implication is that organizations should consider code quality improvements and debt remediation as complementary rather than competing investments.

6.3. Defect Density and Quality Implications

The defect density model confirms that technical debt is not merely a velocity concern but a fundamental quality risk factor. The 31 percent increase in defect density associated with a one-standard-deviation increase in debt density translates directly into increased incident rates, customer impact, and remediation costs. The significant interaction between debt density and code complexity (alpha_4 = 0.22) highlights the particularly dangerous combination of high debt and high complexity, conditions that frequently coexist in legacy systems undergoing incremental modernization.

These findings resonate with the broader software engineering literature on the relationship between structural complexity and defect proneness (McCabe, 1976) while extending it by demonstrating that the complexity-defect relationship is significantly amplified by the presence of technical debt. This amplification effect has not been previously documented in the empirical literature and represents a novel contribution of this study.

6.4. The Economics of Remediation

The cost-benefit analysis provides what is arguably the most actionable finding of this study: structured remediation programs yield substantial positive returns across all debt categories, with median

ROIs ranging from 156 percent for code debt to 437 percent for architectural debt. The median break-even period of 4.7 months for design debt remediation is well within typical corporate planning horizons, suggesting that remediation investments should be economically attractive even under conservative capital allocation criteria.

The finding that architectural remediation yields the highest ROI despite requiring the largest absolute investment resolves a persistent debate in the practitioner community about whether to focus remediation efforts on ”low-hanging fruit” (code-level issues with quick fixes) or ”root causes” (architectural issues with larger remediation costs). The data clearly favor the root-cause approach from an ROI perspective, though the longer break-even period for architectural remediation may create organizational resistance in companies with short-term performance incentives.

The ”experience premium” identified in the survey data, where respondents with remediation experience rated ROI significantly higher than those without experience, suggests the presence of an information asymmetry that may impede optimal investment decisions. Organizations that have never undertaken structured remediation appear to systematically underestimate its benefits, potentially perpetuating a cycle of underinvestment that allows debt to compound unchecked.

6.5. Implications for Theory

The findings contribute to the theoretical understanding of technical debt in several ways. First, the calibration of category-specific interest rates provides empirical grounding for what has been primarily a qualitative distinction in the literature. While Kruchten et al. (2019) and Li et al. (2015) proposed that different debt types have different cost profiles, the specific magnitudes had not been previously estimated from enterprise data. The 2.8:1 ratio of architectural to code debt interest rates provides a quantitative anchor for future theoretical work and simulation models.

Second, the significant interaction effects in both the velocity degradation and defect density models challenge the implicit assumption of linearity that underlies many existing debt quantification tools. SonarQube and similar platforms typically present debt as an additive metric, where each issue contributes independently to the total debt estimate. The interaction terms in the TDIM reveal that debt items do not operate independently but rather interact with contextual factors (code quality, structural complexity) to produce non-linear amplification of their effects. This finding aligns with Martini and Bosch’s (2017) theoretical concept of ”contagious” debt and extends it with quantitative evidence from a broader sample.

Third, the integration of DORA metrics as outcome variables for technical debt research bridges two previously disconnected streams of the software engineering literature. The DORA program has extensively studied the organizational and technical practices that predict delivery performance but has not directly modeled technical debt as a predictor. Conversely, the technical debt literature has focused primarily on identification and measurement, with limited attention to downstream performance consequences measured through standardized frameworks. The TDIM provides a unifying analytical structure that connects these complementary perspectives.

6.6. Comparison with Prior Work

The magnitude of the debt-velocity relationship (beta = -0.41) is broadly consistent with, but somewhat larger than, prior estimates. Besker et al. (2018) reported that developers wasted 23 percent of their time due to technical debt, while the present study estimates a 23 percent velocity reduction at one standard deviation above the mean debt ratio. The consistency of these estimates, derived through different methodologies (self-report surveys vs. objective telemetry), strengthens confidence in the robustness of the finding. Tornhill and Borg's (2022) finding that healthy code was developed at twice the speed of unhealthy code represents a more extreme estimate, likely reflecting their binary classification scheme (healthy vs. unhealthy) compared to the continuous debt ratio used in this study.

The defect density findings align with the broader software quality literature. The mean defect density of 4.8 per KLOC in this sample is consistent with the range of 4 to 8 defects per KLOC reported in industry benchmarks and with the findings of Lomio et al. (2022) in their analysis of Apache Foundation projects. The strong predictive role of cyclomatic complexity (alpha_2 = 0.31) confirms McCabe's (1976) original proposition with modern enterprise data and extends it by documenting the amplifying interaction with technical debt.

7. Practical Implications

The findings of this study yield several actionable implications for technology leaders and software engineering organizations:

- **Implement multi-layer debt measurement.** Organizations should move beyond exclusive reliance on code-level static analysis (e.g., SonarQube) and implement measurement approaches that capture architectural and design debt. The differential interest rates identified in this study (architectural debt compounds 2.8 times faster than code debt) make comprehensive measurement a prerequisite for informed investment decisions.

- **Establish a technical debt budget.** The compound interest model provides a basis for treating technical debt as a financial liability on the technology balance sheet. Organizations should establish explicit debt budgets, analogous to financial debt covenants, that define acceptable debt ratios and trigger remediation actions when thresholds are exceeded. The data in this study suggest that debt ratios exceeding 0.25 (25 percent of development capacity consumed by debt servicing) are associated with significant performance degradation.

- **Prioritize architectural remediation.** Despite higher absolute costs, architectural debt remediation produces the highest ROI (median 437%) and addresses the fastest-compounding category of debt. Organizations should allocate at least 40 percent of their total remediation budget to architectural initiatives, with the remainder distributed across design, code, and test debt.

- **Integrate debt metrics into DORA dashboards.** The strong predictive relationships between debt metrics and DORA performance indicators suggest that technical debt ratio should be

incorporated as a leading indicator in DevOps performance dashboards. A rising debt ratio can serve as an early warning signal for future degradation in deployment frequency, lead time, and change failure rate.

- **Quantify cost of delay.** Technology leaders should routinely calculate the cost of delay using the CostOfDelay formula presented in this study ($C_{\text{delay}} = R_{\text{week}} \cdot P_{\text{delay}} \cdot D$) to translate technical debt from a technical concern into a business case. This reframing is essential for securing investment from non-technical stakeholders.

- **Adopt iterative remediation.** Rather than pursuing large-scale "big bang" remediation programs, the data suggest that iterative approaches integrated into the regular development workflow produce more consistent returns and shorter break-even periods. Projects that allocated 15 to 20 percent of sprint capacity to ongoing debt reduction maintained lower debt ratios and higher velocity than projects that deferred remediation to dedicated cleanup phases.

- **Monitor the quality-debt interaction.** Given the significant interaction between code quality and debt ratio in predicting velocity, organizations should treat simultaneous improvements in quality and debt reduction as synergistic investments. Investing in code review practices, automated testing infrastructure, and continuous integration, capabilities highlighted in the DORA research (DORA, 2023), can reduce the velocity penalty of existing debt while preventing the accumulation of new debt.

- **Use the cost-of-delay framing for executive communication.** The CostOfDelay formula provides a financially intuitive mechanism for communicating technical debt impact to non-technical executives. By expressing debt costs in terms of delayed revenue realization rather than abstract quality metrics, engineering leaders can more effectively compete for budget allocation within enterprise capital planning processes. The median weekly cost of delay of $68,425 per project identified in this study provides a concrete benchmark for initiating such conversations.

- **Benchmark against industry data.** The descriptive statistics presented in this study (mean debt ratio of 0.18, mean deployment frequency of 12.4 per month, mean defect density of 4.8 per KLOC) can serve as industry benchmarks for self-assessment. Organizations whose metrics fall substantially below these benchmarks should investigate whether elevated technical debt is a contributing factor and consider commissioning a formal debt assessment aligned with the TDIM framework.

8. Limitations and Future Research

This study has several limitations that bound the interpretation of its findings and suggest directions for future research. First, the sample of 89 projects, while substantial by the standards of empirical software engineering research, is drawn exclusively from enterprise environments in four industry sectors. The generalizability of the calibrated interest rates and regression coefficients to small and medium enterprises, open-source projects, or emerging technology domains (e.g., machine

learning pipelines, embedded systems) remains an open question. Future research should replicate this analysis across more diverse organizational contexts.

Second, the reliance on SonarQube as the primary source of debt quantification introduces tool-specific measurement bias. SonarQube excels at detecting code-level issues but has known limitations in identifying architectural and design debt (Saarimaki et al., 2019). While the study mitigated this limitation by incorporating architectural debt assessments from the survey data and supplementary code analysis, a fully automated multi-level debt measurement approach would strengthen future iterations of the TDIM.

Third, the cross-sectional survey data limit causal inference. While the longitudinal project telemetry supports directional interpretation (debt accumulation precedes velocity degradation), the survey component captures associations at a single point in time. A longitudinal panel survey design, tracking the same respondents over multiple measurement points, would provide stronger evidence for the causal mechanisms proposed in the TDIM.

Fourth, the cost-benefit analysis relies on organization-reported cost data that may be subject to estimation error and reporting bias. Organizations may overestimate remediation costs (due to the inclusion of opportunity costs) or underestimate them (due to the exclusion of coordination overhead). Future research should employ more rigorous cost accounting methodologies, potentially drawing on activity-based costing frameworks from the management accounting literature.

Fifth, the study does not address the emerging challenge of AI-generated code and its impact on technical debt dynamics. As organizations increasingly adopt large language models for code generation, the accumulation patterns and quality characteristics of AI-generated technical debt may differ substantially from human-generated debt. Forrester (2024) has warned of a ”tech debt tsunami” fueled by rapid AI adoption, a concern that warrants dedicated empirical investigation.

Sixth, the study does not account for the heterogeneity of development methodologies across the sampled projects. While all 89 projects employed some form of agile methodology, the specific implementation varied from Scrum to Kanban to SAFe, and these methodological differences may moderate the relationship between debt accumulation and delivery performance. Projects employing explicit work-in-progress limits or dedicated capacity for technical debt remediation may exhibit different accumulation trajectories than those without such structural safeguards.

Future research should also explore the organizational and cultural moderators of the debt-performance relationship. The SPACE framework's inclusion of satisfaction and well-being dimensions suggests that developer experience may mediate or moderate the impact of technical debt on delivery outcomes. Investigating how organizational culture, psychological safety, and team autonomy influence the effectiveness of debt remediation programs would contribute to a more complete understanding of the sociotechnical dynamics of technical debt management.

9. Conclusion

This study set out to bridge the gap between the growing recognition of technical debt as a strategic concern and the limited empirical evidence connecting measurable debt indicators to quantifiable delivery performance outcomes. Through the development and validation of the Technical Debt

Impact Model, the research demonstrates that technical debt operates as a compound interest mechanism, with architectural debt compounding at rates nearly three times higher than code-level debt, and that the resulting accumulation produces statistically significant and practically substantial degradation across all four DORA delivery performance metrics.

The finding that a one-standard-deviation increase in debt ratio corresponds to a 23 percent reduction in delivery velocity and a 31 percent increase in defect density provides technology leaders with concrete, defensible metrics for communicating the cost of inaction to business stakeholders. The cost-benefit analysis further demonstrates that structured remediation programs yield positive returns across all debt categories, with architectural remediation producing the highest returns despite requiring the largest investments.

The TDIM framework, combining the compound interest accumulation function, the velocity degradation model, the defect density prediction model, and the remediation ROI calculation, provides an integrated analytical toolkit that enables evidence-based decision-making about debt management investments. By grounding these calculations in data from 89 enterprise projects and the perspectives of 412 engineering leaders, the framework balances theoretical rigor with practical applicability.

As software systems continue to grow in scale and strategic importance, the economic consequences of unmanaged technical debt will only intensify. The challenge for the field is not whether to address technical debt but how to do so efficiently, with confidence that remediation investments will yield positive returns. This study offers a step toward that goal by providing the empirical evidence and analytical tools needed to transform technical debt management from an ad hoc activity into a disciplined engineering and financial practice.

The compound interest model, the velocity degradation function, and the remediation ROI framework collectively provide a language for communicating technical debt costs in terms that resonate with both engineering teams and business stakeholders. As the industry confronts the dual pressures of accelerating delivery expectations and aging codebases, the ability to quantify, prioritize, and economically justify debt remediation investments will increasingly distinguish high-performing technology organizations from those trapped in the vicious cycle of accumulating debt and declining productivity.

References

1. Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., & Avgeriou, P. (2015). The financial aspect of managing technical debt: A systematic literature review. Information and Software Technology, 64, 52–73. https://doi.org/10.1016/j.infsof.2015.04.001

2. Avgeriou, P., Kruchten, P., Ozkaya, I., & Seaman, C. (2016). Managing technical debt in software engineering (Dagstuhl Seminar 16162). Dagstuhl Reports, 6(4), 110–138. https://doi.org/10.4230/DagRep.6.4.110

3. Azevedo, D., Cui, S., Gupta, S., & Srivastava, S. (2021). Tech debt: Reclaiming tech equity. McKinsey & Company. https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/tech-debt-reclaiming-tech-equity

4. Besker, T., Martini, A., & Bosch, J. (2018). Technical debt cripples software developer productivity: A longitudinal study on developers' daily software development work. In Proceedings of the 2018 International Conference on Technical Debt (pp. 105–114). ACM. https://doi.org/10.1145/3194164.3194178

5. Besker, T., Martini, A., & Bosch, J. (2019). Software developer productivity loss due to technical debt: A replication and extension study examining developers' development work. Journal of Systems and Software, 156, 41–61. https://doi.org/10.1016/j.jss.2019.06.004

6. Creswell, J. W., & Creswell, J. D. (2018). Research design: Qualitative, quantitative, and mixed methods approaches (5th ed.). SAGE Publications.

7. Cunningham, W. (1992). The WyCash portfolio management system. In Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (pp. 29–30). ACM.

8. Digkas, G., Lungu, M., Chatzigeorgiou, A., & Avgeriou, P. (2017). The evolution of technical debt in the Apache ecosystem. In Software Architecture: 11th European Conference, ECSA 2017, Proceedings (pp. 51–66). Springer. https://doi.org/10.1007/978-3-319-65831-5_4

9. DORA. (2023). Accelerate State of DevOps Report 2023. Google Cloud. https://dora.dev/research/2023/dora-report/

10. Ernst, N. A., Bellomo, S., Ozkaya, I., Nord, R. L., & Gorton, I. (2015). Measure it? Manage it? Ignore it? Software practitioners and technical debt. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (pp. 50–60). ACM. https://doi.org/10.1145/2786805.2786848

11. Forrester. (2022). The Forrester guide to technical debt (Report No. RES177366). Forrester Research.

12. Forrester. (2024). Role connection: The tangled web of technical debt (Report No. RES189934). Forrester Research.

13. Forsgren, N., Humble, J., & Kim, G. (2018). Accelerate: The science of lean software and DevOps: Building and scaling high performing technology organizations. IT Revolution Press.

14. Forsgren, N., Storey, M.-A., Maddila, C., Zimmermann, T., Houck, B., & Butler, J. (2021). The SPACE of developer productivity: There's more to it than you think. Queue, 19(1), 20–48. https://doi.org/10.1145/3454122.3454124

15. Gartner. (2022). Manage technology debt to create technology wealth (Document ID: G00766508). Gartner, Inc.

16. Guaman, D., Alejandro-Quezada Sarmiento, P., Barba-Guaman, L., Cabrera, P., & Enciso, L. (2017). SonarQube as a tool to identify software metrics and technical debt in the source code through static analysis. In Proceedings of the 2017 7th International Workshop on Computer Science and Engineering (pp. 171–175). WCSE.

17. Kruchten, P., Nord, R., & Ozkaya, I. (2019). Managing technical debt: Reducing friction in software development. Addison-Wesley Professional.

18. Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. Proceedings of the IEEE, 68(9), 1060–1076. https://doi.org/10.1109/PROC.1980.11805

19. Li, Z., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt and its management. Journal of Systems and Software, 101, 193–220. https://doi.org/10.1016/j.jss.2014.12.027

20. Lomio, F., Moreschini, S., & Lenarduzzi, V. (2022). A machine and deep learning analysis among SonarQube rules, product, and process metrics for fault prediction. Empirical Software Engineering, 27, Article 170. https://doi.org/10.1007/s10664-022-10164-z

21. Martini, A., & Bosch, J. (2017). On the interest of architectural technical debt: Uncovering the contagious debt phenomenon. Journal of Software: Evolution and Process, 29(10), e1868. https://doi.org/10.1002/smr.1868

22. Martini, A., Besker, T., & Bosch, J. (2018). Technical debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations. Science of Computer Programming, 163, 42–61. https://doi.org/10.1016/j.scico.2018.03.007

23. McCabe, T. J. (1976). A complexity measure. IEEE Transactions on Software Engineering, SE-2(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

24. Nugroho, A., Visser, J., & Kuipers, T. (2011). An empirical model of technical debt and interest. In Proceedings of the 2nd Workshop on Managing Technical Debt (pp. 1–8). ACM. https://doi.org/10.1145/1985362.1985364

25. Reinertsen, D. G. (2009). The principles of product development flow: Second generation lean product development. Celeritas Publishing.

26. Rios, N., Spinola, R. O., Mendonca, M., & Seaman, C. (2018). The most common causes and effects of technical debt: First results from a global family of industrial surveys. In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (Article 39, pp. 1–10). ACM. https://doi.org/10.1145/3239235.3268917

27. Saarimaki, N., Baldassarre, M. T., Codabux, Z., & Lenarduzzi, V. (2019). On the diffuseness of technical debt items and accuracy of remediation time when using SonarQube. Information and Software Technology, 113, 22–38. https://doi.org/10.1016/j.infsof.2019.05.002

28. Srivastava, S., Trehan, K., Wagle, D., & Wang, J. (2020). Developer velocity: How software excellence fuels business performance. McKinsey & Company. https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/developer-velocity-how-software-excellence-fuels-business-performance

29. Stripe. (2018). The developer coefficient. Stripe, Inc. https://stripe.com/reports/developer-coefficient-2018

30. ThoughtWorks. (2023). Technology Radar, Vol. 28. ThoughtWorks, Inc. https://www.thoughtworks.com/radar

31. Tornhill, A., & Borg, M. (2022). Code red: The business impact of code quality: A quantitative study of 39 proprietary production codebases. In Proceedings of the International Conference on Technical Debt (pp. 1–10). ACM. https://doi.org/10.1145/3524843.3528091

32. Zazworka, N., Vetro, A., Izurieta, C., Wong, S., Cai, Y., Seaman, C., & Shull, F. (2014). Comparing four approaches for technical debt identification. Software Quality Journal, 22(3), 403–426. https://doi.org/10.1007/s11219-013-9200-8