LEVERAGING AI TO REDUCE TECHNICAL DEBT

Vijay Pahuja and Vishal Padh

USA

ABSTRACT

Technical Debt is one of the biggest issues hindering the digital transformation of organizations. Cost of addressing debt has been rising. AI powered tools can overcome problems of traditional tools as they continuously learn and adapt new patterns. They can proactively detect issues, suggest refactoring, and provide insight to areas of improvement in the codebase, pushing for more sustainable software development practices. While AI offers tremendous potential for managing and reducing technical debt, AI based tools come with their own challenges as AI is heavily dependent on the quality and quantity of data on which they are trained. As organizations rely more and more on AI, they may end up with monotonous codebases producing mediocre products as use of AI will lead to skill degradation and affect critical thinking.

KEYWORDS

Technical Debt, AI, Software Engineering, Development Tools

1. INTRODUCTION

Building a successful software product require significant effort, planning and expertise. Many things can go wrong in the way like users did not like the interface, competitive product launched, another innovation disrupted the market. This has led to extensive use iterative development where speed of delivering a feature is often favored over perfection allowing for quick market entry and opportunity for receiving early user feedback. This approach often leads to compromise on the quality of software which inevitably leads to the accumulation of technical debt.

Technical debt is often acceptable in agile/iterative software development with the intent that it will be managed in future sprints. The other reasons adding to the problem are unclear requirements, poor architecture/design, poor code quality, poor testing and vulnerable to security threats. If left unmanaged, technical debt becomes a significant liability, slowing down the speed of development, increases the system complexity, makes maintaining and extending the software more challenging. Southwest Airlines [1], Equifax [3], Capital One [12], FAA [1] and CrowdStrike [13] are some of the examples of who have faced challenges of unchecked technical debt resulting monetary losses, lawsuits and loss of consumer confidence.

To achieve long term success of a software product, it crucial to manage the technical debt. In this regard, Artificial intelligence is emerging as a valuable ally that can proactively help by automating tasks like code reviews, testing, and predictively analyzing any other flaws, enabling teams take corrective action early. Use of AI can help delivering product features faster, leading to a healthier, more robust software product that stands the test of time.

2. TECHNICAL DEBT ACCUMULATION

Technical debt accumulation is contributed by factors known or unknown. When we prioritize feature delivery over code quality, design system for smaller user base to save initial cost we take the technical debt intentionally. When product requirements are unclear, users don’t provide feedback, development teams are not well trained on the technologies being used and lack of product domain knowledge we can incur it unknowingly. To manage technical debt effectively it is imperative that we understand its origins, which can be grouped into several key categories:

- Product Feature Debt: This debt primarily results when we rush development, scope change during development, ignoring user feedback and focus on short term benefits over long term vision. This results in poor design of user interfaces, incomplete or redundant features, and defective implementations. Product debt feature will ultimately give away the initial competitive edge we gain due to speedy feature delivery as it will cause user dissatisfaction.
- Architecture/Design Debt: Architecture/design debt arises when flawed design decisions taken in haste by the development team. When we don’t allow for design to evolve with changing requirement, increased user base, newer technologies, and newer security threats we face difficulty in making changes to software. It eventually leads to having an unscalable and fragile software product that is prone to system break down and unsecure.
- Code Debt: We incurCode debt when the teams follow poor coding practices either to meet tight deadlines or due to lack of training and insufficient testing. It can also happen due when development teams are not clear about architectural design and product vision. I have come across many situations where due to tight deadlines code was release with low degree of testing or a feature was implemented with non-optimal code.
- Testing Debt: One of the most common ways in which we accumulate technical debt is testing. We often ignore testing to meet deadlines resulting in codebases with insufficient coverage, poor test design, and lack of automation. This makes developers tend to avoid refactoring, code cleanup activities and add more technical debt while making any changes.

For sustained healthy products it is essential that we address these root causes by proactive technical debt management. Understanding the different debt type allows us for targeted interventions. It also helps in choosing the tools to improve the software quality and maintainability in the long term.

3. FINANCIAL BURDEN OF TECHNICAL DEBT

Technical debt significantly impairs both current productivity and future innovation. Ward Cunningham, who coined the term, described it as accumulating "interest" that can ultimately cripple entire software engineering organizations [7]. It impacts software projects in multiple ways:

- Diminished Developer Productivity: According to Stripe's 2018 "Developer Coefficient" report, developers worldwide dedicate an average of 13.5 hours each week to resolving technical debt [8]. Developers spend this time on rework, debugging, navigating bad code, and sometimes rewriting whole product. This directly impacts feature development and innovation.

- Significant Project-Level Financial Strain:SonarSource's 2023 research based on examinations of 200 projects estimates an annual cost of $306,000 for a project with one million lines of code [10]. This cost is usually incurred to address vulnerabilities, architectural flaws, improve codebase with automated tests and software upgrades.
- Escalating Long-Term Expenditures: The financial impact of technical debt is not static but rather compounds over time. SonarSource research projects that the cumulative expenses for a million-line project can reach $1.5 million over five years [10].
- Opportunity Costs and Innovation Deficit: Stripe estimates that companies globally lose approximately $85 billion annually due to developer time spent on maintenance tasks like debugging and refactoring [8]. This time that could be invested in developing new functionalities, exploring emerging markets, or fostering innovation.
- Strategic and Architectural Constraints: J.P. Morgan's 2022 analysis highlights that technical debt is not just due to code-level, it can be due to suboptimal system architecture as well [11]. Not only this debt hinders the adaptability to evolving business demands and technological advancements, it can reduce the lifespan of a software and make it vulnerable to software breaches.
- Broad Economic Impact of Poor Software Quality: The Consortium for Information & Software Quality (CISQ) 2022 report estimates the total cost of poor software quality in the US alone at $2.41 trillion in 2022 [12]. Stripe's research also estimated the loss at approximately $300 billion annually in global GDP, demonstrating the far-reaching economic impact of unmanaged technical [8]. Y2K bug forced governments and businesses cost an estimated amount of $300 billion to $600 billion [9], diverting capital and resources that could have been otherwise utilized for innovation and growth-oriented projects.

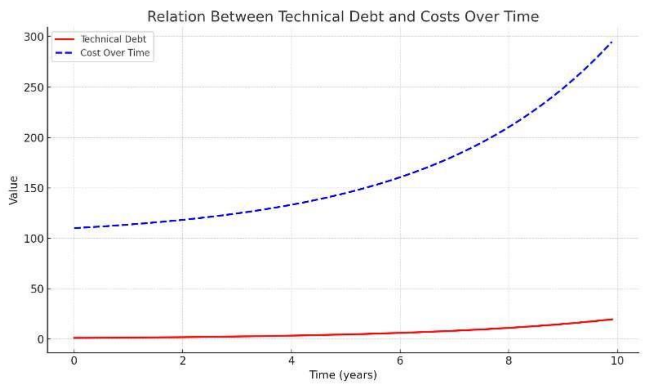

Diagram 1. Relation Between Technical Debt and Costs Over Time

Collectively, these data points paint a clear picture that it is not a minor problem. If left alone, not only it will become significant liability, it is going to be even bigger financial. Proactive and systematic technical debt management is therefore must be made a fundamental requirement to ensure success of software project, its long-term financial stability.

4. THE ADVANTAGES OF TECHNICAL DEBT REMEDIATION

No matter what we do, the technical debt builds up during software development, the question is if we want to keep accumulating or make conscious effort of addressing it and make the software product financially viable for very long term. The benefit of addressing technical debt is not only financial though, it helps the entire software development lifecycle in many ways significantly improving business outcomes. The key benefits can include:

- Accelerated Development Velocity: Stripe's report [8], it can save an average of 13.5 hours per week to address technical debt. While it is generally justified by business to kick the can of technical debt down the road, resolving code-level technical debt through code simplification and enhanced maintainability brings efficiency in the software development. Which ultimately help with delivering the product features faster and lesser defects.
- Elevated Code Quality and Product Reliability: When we keep the focus on quality of codebase by refactoring and any other code optimization techniques, we deliver product with reduced incidents of defects. This is achieved because we end up having testable code. The result is a dependable product. In the projects I have been associated with, my observation is that a quality codebase has lower density of defects resulting in lower expenditure associated with defect resolution. This allows us to invest more in other areas of software product.
- Unleashed Innovation Potential: A well-designed software with clean and testable code base helps us with making larger changes without fear of unknown issues otherwise. This provides us with an environment for innovation and facilitates seamless integration of novel features, encourages experimentation with cutting-edge technologies. A large portion of $2.41 trillion [12] can be spent on new products as result of overall saving. Even at project level an estimated saving of $306,000 [10] can achieved.
- Reduced Long-Term Maintenance Expenditures: When we proactively manage technical debt, the resulting code base easy to manage and costs less to maintain. This codebase will have reduced defect density, simplified code architecture, and decent documentation allowing us to easily patch and upgrade software.
- Enhanced Team Morale and Productivity: We feel very proud when we have a software product with codebase which is well designed with good documentation and focused product strategy. It is also easy to work with reducing developers' frustration. This fosters a sense of professional accomplishment and improves team morale, job satisfaction, and overall productivity.
- Improved Scalability and Performance Characteristics: During initial phases of development some of the areas we deliberately ignore are database optimization, caching, asynchronous processing, and modular architecture. This helps in reducing initial cost and allows quick feedback to check viability of the product. When we address these architectural debt areas as products mature it helps us to improve scalability and performance needed for accommodating increased user loads, expanding data volumes, and facilitating future growth.
- Reinforced Security Posture: When we consistently address security vulnerabilities, we get well designed software product with fortified software security that can mitigate the potential for cyberattacks and data breaches.
- Elevated Customer Satisfaction Metrics: The effect of reduced defects, enhanced performance, timely delivery of new features, and a more streamlined user experience culminates in heightened customer satisfaction.

Table 1. Impact of Technical Debt Remediation

Advantage	Estimated Benefit	Source/Study	Impact
Time Saved via Technical Debt Reduction	13.5 hours per week per developer	Stripe, 2018 [16]	Increases feature development velocity
Defect Density Reduction	40-60% decrease in defect rates	SonarSource, 2023 [15]	Reduces maintenance costs and improves reliability
Innovation Potential Unlocked	$85 billion gained globally per year	Stripe, 2018 [16]	Frees up development resources for innovation
Annual Maintenance Cost Savings (1M LOC)	$306,000 per project annually	SonarSource, 2023 [15]	Reduced costs via proactive refactoring
Developer Productivity Gains	20-30% increase in coding efficiency	Vega, M. (2022), 2022 [18]	Improves overall software delivery speed

The table above summarizes the impact of the addressing technical debt. Despite clear advantages for engineering teams, addressing the technical debt is challenging to explain and justify the investment to business stakeholders.

5. TAMING TECHNICAL DEBT

Even though organizations and development teams know that it is vital to address technical debt for long-term software sustainability and innovation, the topic of technical debt triggers fear. The factors that make us think that reducing the technical debt is difficult are:

- Resource Investment: If we have accumulated considerable technical debt its remediation needs substantial time and budget. This will take away focus from feature releases in short term and create tension with immediate deadlines.
- Instability: Refactoring or architectural changes carry risk of regressions and system instability. If we have accumulated testing debt as well then lack of strong automated testing requires us to think about the unintended consequences. If we have faced negative experiences in past with refactoring then we have a tendency of avoiding to start refactoring.
- Ownership and Prioritization: Often times there is no one to take the responsibility for technical debt and it is believed that engineers in the development team are the sole owners. Unclear prioritization results in inaction, as teams struggle to define ownership.
- Skill and Knowledge: In order to remediate the technical debt, the development team must know the business domain, architecture, technologies involved, the codebase, and current test suites. If team lacks knowledge and skills then they won’t be able to take effective action. This can be a major issue with a legacy product or when a team has high turnover.
- Organizational Culture: Many organizations are trying to promote startup culture in teams when working on a new idea. This helps them to evaluate if new ideas are feasible or not. However, they fail to evolve. While working on new ideas teams usually are asked to make lowest possible investment in all aspects of software development and hence, they accumulate technical debt immediately. If organization don’t build a culture where a successful idea is evolved in mature product and manage

the technical debt, these ideated software products with technical debt are sooner or later going to have issues.

If organizations acknowledge these challenges and adopt strategies that promote a proactive and sustainable approach to software health. We also need to build a culture of understanding that the technical debt management is not a one-time project, but a continuous, strategic process integrated into the development lifecycle.

6. TOOLS FOR TECHNICAL DEBT REDUCTION

Today there are many software tools available to indicate technical debt while it is being accumulated. These tools can be categorized by their primary function:

- Diagnostic Code Assessment (Code Analysis & Static Analysis): Tools that we use in this category includes Linters (for stylistic and programmatic checks), SAST tools (for security and architectural vulnerability detection), and Code Complexity Analyzers (for quantifying and visualizing code complexity). We use these tools identify potential issues right when it is checked in to source control. This will alert the development team for violations of standards, security risks, and complexity hotspots, fixing it immediately [10][17].
- Code Remediation and Optimization (Refactoring & IDE Features): Many modern tools have features that can automate refactoring or assist with suggestion of better code implementation. These tools can help identifying sub-optimal code allowing team to address issue early in development cycle [6][16].
- Quality Assurance and Stability (Testing & Quality Assurance): Automated Testing Frameworks, Code Coverage Tools, and Performance Monitoring & Profiling tools ensure code stability and quality during debt reduction. Utilizing these tools brings confidence to refactoring activity as they help in verifying functionality, measuring test coverage, and identifying performance bottlenecks. There is also an opportunity of integrating these tools in code build pipelines helping in avoiding accumulation of technical debt [10][17].
- Knowledge Preservation and Debt Prevention (Knowledge Sharing & Documentation): Tools like Wiki & Documentation Platforms and Knowledge Sharing Platforms are crucial to manage knowledge base. This helps with on boarding new members to gain understanding of the product and codebase. Documenting decision taken during different stages of software development helps during technical debt remediation [8][17].

Table 2. Pros and Cons of Traditional Tools

Tool	Pros	Cons
SonarQube	Provides detailed code quality analysis and identifies technical debt.	Requires setup and configuration; may have a learning curve.
JIRA	Helps track and prioritize technical debt tasks effectively.	Can become complex to manage for large projects.
Refactoring Tools	Simplifies code improvement and reduces complexity.	May require significant time investment for large-scale refactoring.
Static Code Analysis Tools	Detects code issues early, preventing accumulation of debt.	May generate false positives, requiring manual review.
Automated Testing Tools	Ensures code reliability and reduces defect-related debt.	Initial setup can be timeconsuming; requires maintenance.

In essence, these tools provide us with a comprehensive toolkit for diagnosing, remediating, ensuring the quality of codebase. While invaluable, these traditional tools for technical debt reduction are not without the limitations:

- Automated Analysis Limitations: These tools primarily catch the syntax and style issue and lack the capabilities to surface deeper code flaws. They are prone to false positives and negatives, which requires human validations. Additionally, these tools don't have understanding of business domain, which can lead them to misinterpret intentional design choices [20].
- Automated Refactoring Challenges: These tools are not very effective in refactoring of complex codebases which have evolved to their state over a long period of time. When refactoring done via the automated tools, there is high risk that they will introduce bugs requiring careful testing. The tools are not capable of addressing broader design or architectural debt [6].
- Reliance on Test Quality: The effectiveness of tools is limited by the quality and coverage of the test. Garbage in garbage out, if the tests are not of high quality the refactoring is not guaranteed. These tools also struggle with technical debt issues around performance and usability [17].
- Human & Process Limitations: Almost all tools that are available today require humans or processes. The teams need to adopt the tools, once they are adopted, they must be embedded in the process of software development so that they are consistently used. Tools can't remediate any issues when the recommendations are ignored or overridden [14].
- Documentation & Knowledge Sharing Limits: Documentation tools are useless if content is outdated or incomplete. We must actively update and share information on knowledge platforms else it will be useless [8].

While traditional tools have been useful certain facets of technical debt mitigation, their inherent limitations are obvious. The constraints in these tools underscore the need for more evolved and intelligent approaches. Fortunately, the advancements of AI present a compelling paradigm shift, offering a promising trajectory to address these limitations and reshape our strategies for achieving and maintaining robust codebase integrity.

7. AI TO RESCUE

We see that in near future AI-powered solutions are going to emerge and offer us with enhanced capabilities in technical debt management. They hold promise in several key areas:

- Continuous Learning and Improvement vs. Static Analysis Limitations: AI tools would be able to learn from vast datasets of code. They can be analysing these datasets all the time learning new rules, updating existing or delete obsolete. This is going to be huge advantage as traditional static analysis rely on fixed, potentially outdated rules, we also have opportunity to feed them datasets for learning that can provide better context [15].
- Context-Aware Analysis vs. Context Blindness: AI tools are capable of analysing entire code repositories within the organization. This will help us in building rules that make sense for organization based on business domain, architectural and design decisions taken across the organization, testing tools and strategies adopted. This will provide a holistic system understanding, unlike traditional tools analysing code in isolation [18].
- Adaptive Rule Learning vs. Fixed Rule Sets: As we know software development field is very dynamic field; the practices continuously change. There are new technologies emerging frequently. AI tools can learn and adapt to these new coding standards, best practices, and emerging technologies and automatically update their rules. These AI

based tools can then provide suggestion that will help adopting current industry trends in no time [15][17].

- Proactive Issue Detection vs. Reactive Analysis: Current tools analyze the code when we submit the code for review, which usually triggers actions to perform code analysis. AI can help us shifting from this reactive analysis to proactive issue detection by analyzing code changes in real-time. They can identify patterns already learned and provide us with immediate feedback like we are pair programming. This proactive approach will help us preventing debt accumulation [16].
- Enhanced Automated Refactoring vs. Limited Automation Scope: Not only AIdriven tools can perform complex refactoring compared to IDE tools, they can help us with finding larger issues, which can help with improved performance and scalability of the software product. A lot of this work can be done by automation, which involves not only code refactoring, but any design changes, test automation or any other area of codebase [10] [15].
- Improved Code Reviews vs. Human Variability: Code reviews today are human dependent. Once the code is checked in for code review it has to wait for peer review before being accepted. If AI is able to learn from multiple code repositories, it can help with code reviews, ensuring adherence to coding standards more reliably than human reviewers alone. Code reviews also vary from person to person, whereas AI tools can provide uniform rules, making code base easy to understand [17].
- Performance Monitoring and Optimization vs. Testing Performance Debt:AIpowered performance monitoring tools can analyze runtime data to identify performance bottlenecks and suggest optimizations which are often missed by static analysis and traditional testing [10].
- Facilitating Collaboration and Knowledge Sharing vs. Human Factor Limitations: AI tools can enhance our collaboration by providing platforms for sharing best practices, documenting common issues, and suggesting AI-driven solutions. AI tools are capable of doing a very good job of creating documentation of existing codebase or any new code being added, which can reduce training time we have to put when onboarding new developers on the team [8].

Table 3. Quick Comparison Traditional vs AI-Powered Tools

Area of Comparison	**Traditional Tools (e.g., Static Analysis)**	**AI-Powered Tools**
Learning and Improvement	Rely on fixed, potentially outdated rules	Continuously learns and updates rules from vast code data.
Context Awareness	Analyze code in isolation (Context Blindness)	Analyzes organization-wide code for holistic, context-aware rules.
Rule Adaptation	Fixed rule sets	Automatically adapts to new standards and technologies for timely suggestions.
Issue Detection Timing	Reactive analysis (upon code submission)	Proactively detects issues in real-time with immediate feedback.
Automated Refactoring Scope	Limited automation scope (e.g., IDE tools)	Performs complex, wider-scope automation for performance and scalability.
Code Review Consistency	Human dependent, variability between reviewers	Provides more reliable and uniform code review adherence to standards.
Performance Monitoring & Optimization	Limited (Testing Performance Debt)	Analyzes runtime data to identify and suggest performance optimizations.

Area of Comparison	Traditional Tools (e.g., Static Analysis)	AI-Powered Tools
Collaboration & Knowledge Sharing	Human factor limitations	Enhances collaboration with platforms for sharing, documentation, and solutions.

With significant advances in Generative AI, the usage of AI powered tools has significantly increased. GitHub Copilot research on impact of code quality [22] found that 85% of developers felt that quality of code was better when using GitHub Copilot and code reviews being completed 15% faster. With more time available the engineering teams can focus more on architecture, design and better requirement understanding, which will further help with tech debt reduction.

AI powered tools provide advanced capabilities that can actively help preventing the technical debt. However, we need to carefully evaluate these tools before implementing them in practice.

8. Important Considerations for AI-Powered Tools

While promising, AI-powered tools for technical debt reduction are still in their infancy and their usage without any supervision can cause problems:

- Data Dependency and Bias [10][18]: AI models are dependent on the dataset on which they learn. This may make these tools biased. If the datasets are not wholistic then we may end up with ineffective tools.
- Explainability and Trust [14]: "Black box" AI recommendations without clear explanations can make it difficult for us to trust and adopt the suggestions. If the recommendations are not explained well by the tools, then we as a tendency are going to reject them.
- Automation Over-reliance [17]: AI tools bring the risks of neglecting crucial human oversight and critical judgment. We think AI should augment our expertise and not replace it. Relying blindly on these tools may cause issues that are found very late in the process making troubleshooting and correction of difficult.
- Organizational Implications: Integration of AI powered tools into organization requires shift in culture. We must ensure that the developers are trained to adapt these tools and provide assurance to engineering teams that they are not a replacement as significant oversight is required for its adaption. Engineering leadership should foster transparency, they should not assume that AI can replace human engineers as these tools are not aware of nuances of business, legacy systems in use, skills and knowledge possessed by engineers. There should be processes defined so that any improper usage of AI is detected early, protecting organization from any legal problems.
- Ethical Implications [10]: There is ongoing debate on AI integration to understand, to what extent the usage is ethical. Who owns AI generated code, which may be based on other codebases that the tools used to learn? Will developers lose autonomy? Do AIdriven tools have biases? To address these questions, we think that the organizations must define policies related to ethical use of AI. This will help engineering teams make decisions on if the suggestion by AI tools can be utilized without any broader implications on organization.

Despite these concerns, AI-powered tools represent a significant advancement in technical debt management. They offer a path to overcome traditional tool limitations. However, we must take a balanced approach of integrating AI with human expertise, critical thinking, and a strong code quality culture to realize its full potential.

9. Future Improvements

As codebases of any system grows, the legacy tools will start becoming more and more entrenched. To keep up, we need more research and enhancements in below areas.

- AI-Powered code refactoring tools: More work needs to be done in developing tools that are AI-powered and are good at code refactoring. Tools can be taught to automatically detect issues with large codebases and suggest and/or apply refactoring changes. Points to focus on in the new tools – o Easy modularization – tools can help to break functions/methods into smaller parts for better handling

  - Detect bad practices/anti-patterns – detection of bad practices and suggesting a fix will make it more likely that the developer accepts and tries to incorporate the suggested good practice in code.
  - Auto detection of unused code/feature segments to keep codebase clean

- Smart dependency management: More work needs to be done in the area of dependency management. It's a dynamic and highly integrated programming world today. Everyone uses different APIs or endpoints. These APIs are constantly evolving themselves as new risks and vulnerabilities are uncovered. Points to focus on – oVersion conflicts – any dependent code can easily run into version conflicts if not upgraded timely and appropriately. AI can help uncover these issues and suggest fixes.

  - Replace vulnerable components – as soon as a new version is released for a component, AI can help to replace all vulnerable code with the new versions available.

- Architecture enhancement: Newer models can be developed and applied to analyse and determine the best architecture for any requested system or subsystem. This will not only help maintainability, but might also help performance and scalability.
- AI driven testing: AI can help generate better and deeper test case code. Not only for unit testing, but also for integration and regression testing. This will give higher test coverage, reduce the number of bugs leaking into production, and hence reduce the overall cost.

  Sure enough, there would be way more topics like predictive maintenance for codebases, natural language interfaces for legacy systems, integration with CI/CD for real-time recommendations etc., but we think that the ones mentioned above need most immediate attention.

10. Conclusion

Traditional tools like linters, SAST, and testing frameworks we have been using are valuable. However, they have limitations in semantic understanding, context awareness, and automation complexity. Emerging AI-powered tools can overcome these limitations. continuously learning, context-sensitive analysis, and adaptive rule derivation, making AI based tools invaluable assets in tackling technical debt.

However, AI tools come with their own challenges. These challenges include data dependency, explainability, trust, and the potential for over-reliance and bias. It is imperative that a balanced approach is taken that combines AI's strengths with human expertise, critical thinking, and a strong code quality culture so that tech debt can be managed effectively while building sustainable software systems.

Managing technical debt with use of AI tools, can lead to a better codebase. The use of AI tools can make product development cost effective without sacrificing speed to market, performance, and security. The initial cost of managing technical debt leads to higher productivity, customer satisfaction and team morale

REFERENCES

[1] Kornblith, Aaron. "Lessons from the Runway: How Southwest's System Crash Illuminates Healthcare's Technical Debt Problem." UCSF Synapse.

[2] Devico.io. "Equifax Data Breach of 2017: Unpatched Software Vulnerabilities and Security Debt." Devico.io.

[3] Rackspace Technology Staff. "Capital One Data Breach: Two Security Controls You Should Review." Rackspace.

[4] Artocain.com. "The FAA Outage as a Resiliency Test: Technical Debt in Outdated Technology." Artocain.com.

[5] Sekhon, Kirat. "CIOs Must Tackle the Global Tech Debt Crisis." CIO Dive.

[6] M. Fowler, "Technical Debt," 2003. [Online]. https://martinfowler.com/bliki/TechnicalDebt.html

[7] Cunningham, W. (1992). The WyCash Portfolio Management System. OOPSLA'92 Workshop on Directions in Object-Oriented Programming.

[8] Stripe. (2018). The Developer Coefficient. Stripe Report. stripe.com

[9] National Museum of American History, "Y2K," [Online]. https://americanhistory.si.edu/collections/object-groups/y2k

[10] SonarSource. (2023). Technical Debt: The Silent Drag on Software Development. SonarSource Research.

[11] Myrna Vega. (2022). Insights on Technical Debt. J.P. Morgan Research.

[12] Herb Krasner. (2022). The Cost of Poor Software Quality in the US: A 2022 Report. CISQ Report.

[13] McKinsey & Company. The economic impact of technical debt. McKinsey & Company Report.

[14] Curtis, B. (2012). Conflicting priorities. ACM Queue, 10(10), 10.

[15] Fowler, M. (2020). Technical Debt Quarterly. martinfowler.com

[16] Kruchten, P. (2016). The rational unified process made easy: a practitioner's guide to the RUP. Addison-Wesley Professional.

[17] Li, Z. (2015). Software Development and Professional Practice. Springer.

[18] Li, Z., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt. Journal of Systems and Software, 106, 1-18.

[19] Slinger, J., Erlikh, I., &Bankevich, L. (2019). Technical Debt in Practice: How Do Developers Experience It? 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).

[20] Tom, G., Aurum, A., & Petersen, K. (2013). Technical debt in practice: a grounded theory study of industry perceptions. Journal of Systems and Software, 86(1), 183-198.

[21] Zazworka, N., Shaw, M., & Bachmann, F. (2014). Technical debt: understanding the metaphor. Information and Software Technology, 56(12), 1327-1342.

[22] Research: Quantifying GitHub Copilot’s impact on code quality. Available: https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-codequality/