Uluslararası Sürdürülebilir Mühendislik ve Teknoloji Dergisi

International Journal of Sustainable Engineering and Technology

ISSN: 2618-6055 / Cilt:9, Sayı:1, (2025), 25 – 40

Araştırma Makalesi – Research Article

TECHNICAL DEBT ASSESSMENT IN OPEN SOURCE APPLICATIONS USING STATIC CODE ANALYSIS

*Rafet GÖZBAŞI[1], Kökten Ulaş BİRANT[2]

[1]Dokuz Eylül University, The Graduate School of Natural and Applied Science, Department of Computer Engineering, İzmir
[2]Dokuz Eylül University, Faculty of Engineering, Department of Computer Engineering, İzmir

(Geliş/Received: 10.05.2025, Kabul/Accepted: 02.06.2025, Yayınlanma/Published: 30.06.2025)

ABSTRACT

The management of technical debt is critical to the sustainability of software quality throughout the evolution of software systems. This study investigates how technical debt levels change across multiple versions of four widely used open source software projects, nopCommerce, OrchardCore, RavenDB, and ShareX. Versions 30, 28, 20, and 15 of these projects, respectively, were systematically analyzed using the NDepend static analysis tool to measure technical debt levels. The results reveal that technical debt tends to accumulate, decrease, and stabilize. While nopCommerce exhibits significant increases in technical debt after major version migrations, OrchardCore maintains low levels, demonstrating the effectiveness of modular architecture in debt management. RavenDB exhibits a limited but increasing debt profile, while ShareX tends to reduce its debt level over time. These findings highlight the importance of continuously monitoring and proactively managing technical debt, especially during major structural changes. The study presents empirical findings on the evolution of technical debt in open source projects and demonstrates the value of static analysis tools such as NDepend to software quality management.

Keywords: Static code analysis, Open source applications, Static analysis tools, Technical debt

AÇIK KAYNAK KODLU UYGULAMALARDA STATİK KOD ANALİZİ İLE TEKNİK BORÇ DEĞERLENDİRMESİ

ÖZ

Teknik borcun yönetimi, yazılım sistemlerinin evrimi boyunca yazılım kalitesinin sürdürülebilirliği açısından kritik bir öneme sahiptir. Bu çalışma, yaygın olarak kullanılan dört açık kaynak yazılım projesi olan nopCommerce, OrchardCore, RavenDB ve ShareX'in çoklu sürümleri boyunca teknik borç seviyelerinin nasıl değiştiğini araştırmaktadır. NDepend statik analiz aracı kullanılarak bu projelerin sırasıyla 30, 28, 20 ve 15 sürümü sistematik olarak analiz edilmiş ve teknik borç seviyeleri ölçülmüştür. Elde edilen sonuçlar, teknik borcun birikim, azalma ve stabilite eğilimleri gösterdiğini ortaya koymuştur. nopCommerce, büyük sürüm geçişleri sonrası teknik borçta belirgin artışlar sergilerken, OrchardCore düşük seviyeleri koruyarak modüler mimarinin borç yönetimindeki etkinliğini göstermiştir. RavenDB, sınırlı ancak artış eğilimli bir borç profili sergilerken, ShareX zamanla borç seviyesini azaltma eğiliminde olmuştur. Bu bulgular, özellikle büyük yapısal değişiklikler sırasında teknik borcun sürekli izlenmesi ve proaktif olarak yönetilmesinin önemine dikkat çekmektedir. Çalışma, açık kaynak projelerde teknik borcun evrimine ilişkin ampirik bulgular sunmakta ve NDepend gibi statik analiz araçlarının yazılım kalitesi yönetimine sağladığı değeri ortaya koymaktadır.

Anahtar kelimeler: Statik kod analizi, Açık kaynak kodlu uygulamalar, Statik analiz araçları, Teknik borç

***Sorumlu Yazar/Corresponding Author:** rafetgozbasi@isparta.edu.tr

1. Introduction

In the software field, issues such as software maintainability, maintaining code quality, and code analysis are becoming increasingly important. Open source software is a type of software where the source code is publicly available and users can review and modify the software. Today, open source software is constantly updated by a large community of developers, but these updates can compromise the integrity of the software and create undesirable changes in quality metrics. Capiluppi and Ramil [1] investigated version management and maintainability issues in open source projects and analyzed the effects of version changes on the evolution of projects. The research findings emphasize the importance of version management for the maintainability of software development processes. Scacchi [2] evaluated the applicability of process improvement practices in open source projects by examining the unique problems that arise in the quality and process management of open source software and the methods for solving these problems. Stol and Fitzgerald [3] have deeply examined the complexities of open source software development processes and the difficulties that arise in managing these processes and have proposed sustainable software development methods. Beller et al. [4] studied the prevalence, configuration patterns, and time evolution of nine different static analysis tools across more than 168,000 open source projects. Configurations were often based on default settings, and developers rarely defined custom rules. This analysis showed that 65% of the alert rules were related to maintainability and 35% were related to functional errors.

Static code analysis is one of the widely used methods to measure software quality with specific metric values. Static code analysis examines the source code without running the software and allows developers to evaluate various metrics such as method complexity, technical debt, number of methods, etc. Ayewah et al. [5] evaluated the effectiveness of static analysis outputs in preventing bugs using the FindBugs tool and emphasized the importance of integrating these tools into software processes. Basutakara and Jayanthi [6] studied the role of static code analysis in improving software security and systematically summarized the methods, tools, and analysis models used for early vulnerability detection. This study stated that static analysis tools analyze the code according to abstract syntax trees, control flow graphs, and call graphs. Sultanow et al. [7] presented a machine learning-based analysis approach that aims to detect invisible defects in large-scale general software for cases where classical static code analysis falls short.

Yeboah and Popoola [8] analyzed 1600 user reviews of SonarQube, PMD, Checkstyle, and FindBugs using problem modeling to uncover users' concerns and preferences about static analysis tools. Lenarduzzi et al. [9] compared six popular static analysis tools (SonarQube, Better Code Hub, Coverity Scan, FindBugs, PMD, and Checkstyle) on 47 Java projects and examined their detection capability, overlap, and accuracy. Nachtigall et al. [10] studied the usability issues of static analysis tools around the concept of “explainability” and identified six main challenges that prevent developers from communicating effectively with these tools. In this study, seven industrial and seven academic static analysis tools were evaluated for their responses to these challenges.

NDepend is a powerful static analysis tool widely used in the .NET ecosystem. The tool offers advanced metrics for calculating and tracking technical debt. Avgeriou and Taibi [11] evaluated NDepend among the tools that focus on technical debt measurement, stating that it stands out with its comprehensive metric support and code query capabilities. Ernst et al. [12] stated that tools such as NDepend report not only design-level but also code-level rule violations, so developers should be careful to distinguish between design-related debt and routine code errors.

Coulin et al. [13] systematically examined the metrics used to measure architectural quality and revealed the relationship between these metrics and quality attributes such as maintainability, extensibility, and performance. Debbarma et al. [14] evaluated widely used complexity metrics and studied the contribution of static analysis-based metrics to software testing processes. Ludwig et al. [15] analyzed technical debt levels using code metrics such as architectural complexity obtained with the Understand tool, and showed that certain code components accumulate debt over time. Hernandez-Gonzalez et al. [16] discussed the role of fundamental metrics such as coupling, dependency, complexity, testability, and reusability in the context of software engineering in their systematic review. Nuñez-Varela et al. [17] analyzed 226 studies on source code metrics and identified trends in this area.

ISSN: 2618-6055 / 9, (1), 25 – 40, 2025
DOI:10.62301/usmtd.1696804

Cunningham [18] first defined the concept of technical debt and drew attention to the problems that short-term decisions can create in the long term. The role and effects of technical debt in the software development process are conceptually expressed in this study. Kruchten, Nord, and Ozkaya [19] thoroughly examined the theoretical background and practical applications of the concept of technical debt and presented a comprehensive study emphasizing the importance of technical debt in software engineering processes. Alves et al. [20] examined technical debt indicators and their reflections on software projects and proposed methods for measuring and monitoring technical debt. Ernst et al. [21] analyzed the effectiveness of static analysis tools in identifying technical debt, supporting the importance of these tools in software engineering processes with empirical data. Fontana et al. [22] stated that the tools provide an index that gives an overall assessment of technical debt in a project and explained how to calculate technical debt indices using 5 different tools.

The concept of technical debt is not just a theoretical framework, but a concrete problem that software teams encounter in their daily practice. Holvitie et al [23] stated that with the widespread use of the agile development approach, teams are exposed to an increase in technical debt due to reasons such as frequently changing customer requirements, time pressure, insufficient test coverage, incomplete documentation and resource constraints. This accumulation leads to a decrease in software quality, increased maintenance costs, and structural complexities that make it difficult to manage technical debt. Recent studies emphasize that technical debt accumulates not only at the code level, but also at the architecture, testing, documentation, and process levels, and that failure to effectively manage these debts threatens the success of software projects [24].

With the acceleration of digitalization, software quality has become one of the priority agendas of both researchers and developers. Kokol [25] analyzed the publications in this field and revealed that studies on software quality have increased rapidly in recent years and that research focuses especially on topics such as the development of software engineering processes, testing techniques, and error prediction based on machine learning. Although static analysis tools are effective in detecting software errors early, they can lose the trust of developers due to high false alarm rates. Kang, Aw, and Lo [26] found that the "Golden Features" feature set proposed to increase the accuracy of these tools appeared to be more successful than it actually was due to data leakage and duplicate data. It was also stated that more reliable evaluation methods should be developed to ensure label accuracy.

The issue of technical debt is related to the fact that choices made for the sake of short-term gains make the maintenance and development process of the software difficult in the long run. Melo et al. [27] stated that there are still significant gaps in the identification and measurement of technical debt, especially in the requirements phase, and this can negatively affect software quality. Addressing the general conceptual confusion in this field, Junior and Travassos [28] systematically examined the existing literature on the definition, types, causes and management of technical debt and argued that a common understanding model should be created. Finally, Yadav et al. [29] reported that they achieved more successful results in metrics such as accuracy and faulty sample detection than classical methods with the HEHO-CLSTM method they developed to predict software quality with artificial intelligence.

The main research problem addressed in this study is to answer the question of how the concept of technical debt changes across different software versions and how these changes affect software quality. The questions of whether changes across different versions of a software show a certain trend as open source software versions evolve and whether commonalities or differences exist in technical debt changes across different types of software will also be addressed. In this context, studies analyzing version-based technical debt across different software types are limited. This study aims to fill this gap in the literature by comparatively analyzing the software development processes of different application types.

The primary objective of this study is to examine how technical debt evolves across different versions of structurally diverse open source software projects and to evaluate its implications on software quality. To this end, the research focuses on four widely used .NET-based applications (nopCommerce, OrchardCore, RavenDB and ShareX) analyzing a total of 93 versions using the NDepend static code analysis tool. By applying a uniform metric framework, this study enables both intra-project and cross-project comparisons. The resulting insights aim to guide software developers and researchers by identifying how architectural choices and versioning strategies impact debt accumulation. Furthermore,

ISSN: 2618-6055 / 9, (1), 25 – 40, 2025
DOI:10.62301/usmtd.1696804

the study contributes a replicable and systematic methodology for version-based static analysis that enhances the understanding of software maintainability through empirical evidence.

Although there are many studies analyzing technical debt using static code analysis, few have conducted a longitudinal, version-based investigation across multiple types of open source systems with a consistent metric framework. This study differentiates itself by simultaneously analyzing four structurally distinct applications over a total of 93 versions, using a uniform methodology and metric focus via NDepend. Unlike prior studies that often focus on a single system or metric, this research enables both intra-project and inter-project comparisons of technical debt evolution, offering a comprehensive perspective on how software architecture and versioning strategies influence debt accumulation patterns.

2. Material and Method

In this study, static code analysis method is used to analyze the structural changes of open source software over time. This approach provides numerical data about software quality only from the source code without requiring dynamic execution, and provides a strong evaluation ground, especially for comparisons between versions. In this section, technical debt is introduced, the preferred analysis tool is explained, the open source projects examined are defined, and the analysis process is explained in detail step by step.

2.1. Software metrics and technical debt

Honglei et al. [30] defined software metrics as follows: Software metrics give some quantitative descriptions of the attributes and these attributes are extracted from the software product, software development process and related resources. Technical debt is defined as the cost that software has to pay in the long run due to compromises or shortcuts that developers take to achieve some goals [31].

Technical debt was determined both to track the structural evolution of the code and to enable objective comparison across projects. It was measured with the NDepend tool for each implementation and applied equally to each release using the same methodology.

2.2. Open source applications

In this study, four open source and .NET based applications representing different software categories were examined. The versions of the applications were obtained from their official repositories on GitHub, and the criteria of widespread use, active development, having a clear version history and representing different functional areas were taken into account in the selection process of the applications [32-35]. Thus, the analysis results obtained are intended to be both extensible and comparable across software types. In this context, nopCommerce, an e-commerce platform, OrchardCore, a content management system, RavenDB, a document-based NoSQL database, and ShareX, a screenshot-taking and sharing application, were included in the study.

NopCommerce is a well-known e-commerce platform available as free software that allows interested users to open an online store [36]. It has a layered and modular architecture. Orchard Core is an open-source, modular, multi-tenant application framework and CMS for ASP.NET Core [37]. RavenDB is described as a transactional, open-source document database written in .NET and offers a flexible data model designed to meet your needs [38]. ShareX is known as a monolithic desktop application and is described as a free and open source screenshot and screen recording software for Microsoft Windows.[39]. Since all of these applications are open source applications, their development strategies are basically the same. They are all versioned by individuals or communities.

While performing the analysis, 30 versions were included in the analysis for the nopCommerce application, 28 for OrchardCore, 20 for RavenDB, and 15 for ShareX. When selecting the version, compilable versions that contained functionally significant changes were preferred. In this way, it was possible to examine and interpret the software structure changes of the projects in a healthy way over time. All versions were organized under separate folder structures, preserving the source code integrity; analyzes were performed directly on the source files.

In this study, four open source applications were selected to provide functional diversity while maintaining a consistent technical base. All four projects are built on the .NET technology stack, are publicly available under open source licenses, and are accessible via GitHub and provide a structured version history. Moreover, they represent different application domains. This diversity allows for an insightful comparison of how technical debt evolves in software systems with different architectural styles, purposes, and usage contexts.

The number of versions analyzed for each project was determined by both functional relevance and technical feasibility. While our initial goal was to analyze as many versions as possible to capture long-term evolution patterns, certain limitations emerged during the compilation and analysis process. In particular, older versions of some applications caused critical compatibility issues in Visual Studio and could not be compiled successfully. Furthermore, NDepend requires compilable assemblies to perform static code analysis. These conditions were not met due to outdated frameworks, missing dependencies, or deprecated APIs, and therefore these versions could not be included in the analysis. As a result, the number of versions included per application was not uniform, but instead is based on a subset of stable and analyzable versions.

Each selected release was selected based on two key criteria: compilability and architectural or functional importance. To ensure consistency and accuracy in metric calculation, only releases that can be compiled in a modern .NET environment (Visual Studio 2022) without major code changes were included. Priority was also given to releases that introduced significant changes, such as major releases, incremental updates affecting core modules, or structural redesigns. This ensured that the observed technical debt fluctuations were not only measurable, but also meaningfully linked to changes in system design and development decisions.

NopCommerce was specifically chosen for its maturity, commercial adoption, and detailed version history. As one of the most widely used open source eCommerce platforms in the .NET ecosystem, it undergoes regular updates with clear documentation and change logs. Additionally, nopCommerce's migration to ASP.NET Core and extensive use in real-world deployments provide a practical context for interpreting the results, thus increasing the applicability and relevance of the study findings.

2.3. Static code analaysis tools and NDepend

Various static analysis tools have been developed to assess software quality based on code-level metrics. Static analysis tools are widely used to assess maintainability, complexity, and technical debt indicators. Among them, NDepend stands out due to its deep integration with the .NET ecosystem and extensive support for technical debt measurement. Stanković [40] conducted a comparative evaluation of four static analysis tools, SonarCloud, Squore, Sonargraph, and NDepend, for the purpose of identifying technical debt in .NET projects. NDepend stands out because it can identify not only code debt but also architectural and design debt. While most of the tools use the SQALE methodology, NDepend offers more detailed analysis with its own proprietary metrics. In tests conducted on six open source projects, NDepend produced the highest technical debt estimates and detected more extensive issues compared to other tools. In conclusion, although there are differences between the tools, NDepend stands out as a powerful tool for in-depth analysis and architectural assessment.

In the study conducted by Pfeiffer and Lungu [41], while examining how technical debt and sustainability are measured by software tools, 11 popular static analysis tools were examined in detail. One of these tools, NDepend, defines technical debt as the correction period corresponding to the violated rules and calculates it in man-days. The outstanding feature of NDepend is that it offers more than 200 analysis rules in a customizable way with LINQ-based queries. The debt value corresponding to the violation of each rule can be determined by the user. In addition, indicators such as annual interest and technical debt ratio are calculated to score the total debt status of the software. The study reveals that NDepend performs technical debt calculations in a transparent and customizable manner, and in this respect, it is in a rare position in the industry. Despite its high configurability and detailed measurements, the existing literature does not provide direct empirical validation for the accuracy of NDepend's technical debt estimates. The study of Lefever et al. [42] revealed significant variability among tools using similar methodologies, meaning that even advanced tools like NDepend should be used with caution when drawing quantitative conclusions.

Avgeriou et al. [11] identified NDepend as one of the most feature-rich tools in their comparative overview of technical debt quantification platforms, noting its capabilities to calculate principal, interest, and debt ratio using customizable rule definitions. Similarly, Shaukat et al. [43] assessed the extent of architectural and maintainability issues in NDepend by comparing it with specialized analyzers for safety-critical software. Pavlič et al. [44] considered NDepend as a SQALE-based debt estimator alongside SonarQube and Squore, and found it to be compliant with industry standards for technical debt quantification. Furthermore, Saraiva et al. [45] included NDepend in a systematic mapping of TD tools, noting its effectiveness in agile environments and its recognition in previous work on supporting debt detection in evolving systems. These studies collectively confirm the relevance and widespread adoption of NDepend in both academic and industrial settings, while also providing a critical perspective on the challenges of tool-to-tool consistency in measuring algorithmic coverage, reliability, and technical debt. Their findings strengthen the methodological soundness of using NDepend to perform version-based technical debt analysis in .NET-based open source projects in this study.

During the tool selection phase, alternative tools were considered, but most of them were not compatible with .NET projects or did not support advanced technical debt estimation frameworks. Therefore, NDepend was selected based on its methodological depth, platform compatibility, and empirical support in comparative studies.

In this study, NDepend version 2024.2.1 was used and the analyzes were performed on the Visual Studio 2022 IDE in a development environment with Windows 11 operating system. Thanks to the analysis performed, it was possible to make a comparative evaluation between different versions. NDepend was chosen because it provides the level of detailed metrics required by the study and offers the flexibility to perform version-independent analysis. Compared to alternative tools, NDepend’s .NET-focused analytical depth shows higher overlap with the purpose of this study.

2.4. Analysis Process and Method Followed

In this study, static code analysis was performed on different versions of selected open source .NET applications and the obtained technical debt values were used to evaluate the variation between versions. The analysis process was carried out systematically by following the same steps for all applications and the comparability of the results was ensured. The details of this process are explained below.

First, after identifying the applications, stable versions of each application were manually downloaded from the official repository on GitHub. Each version was organized under separate folder structures to avoid confusion and preserve the integrity of the source files.

An independent NDepend project file was created for each version and the same analysis rules were applied to all versions. This is especially important for measurement consistency. The set of metrics used during the analysis was kept constant; only the technical debt metric was focused on. The analysis operations were completed efficiently thanks to the user interface provided by NDepend.

After each version analysis, the obtained technical debt values were exported via HTML formatted reports produced by NDepend. The data was compiled in Microsoft Excel, processed in separate sheets for each application and tabulated comparatively across all versions. Then, line charts were used to visualize the changes between versions. During the visualization process, trends of increasing, decreasing and fluctuation of the technical debt value were highlighted.

Since the code sizes of the applications are quite different from each other, an approach based on ratios rather than direct absolute values was adopted in order to make a fair comparison between the applications. The order of the basic operations performed during the analysis process is given in Figure 1 as a flow chart.

ISSN: 2618-6055 / 9, (1), 25 – 40, 2025
DOI:10.62301/usmtd.1696804

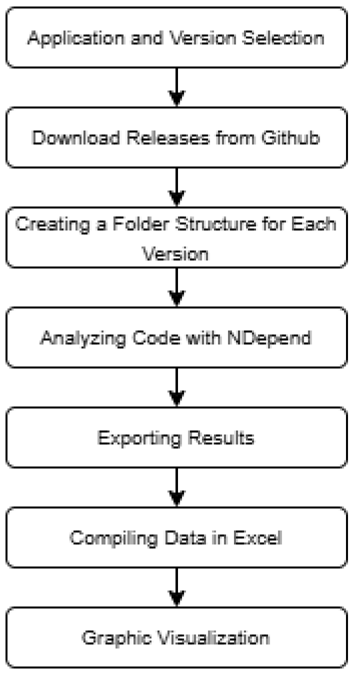
Application and Version Selection
Download Releases from Github
Creating a Folder Structure for Each Version
Analyzing Code with NDepend
Exporting Results
Compiling Data in Excel
Graphic Visualization

Figure 1. Flow chart of the method followed

3. Result and Discussion

In this study, different versions of four open source software projects, namely nopCommerce, OrchardCore, RavenDB and ShareX, are analyzed in terms of technical debt metric using the NDepend tool. The technical debt trend of each project across versions is evaluated both within itself and in comparison with other applications.

The technical debt value for each version for nopCommerce is given in Table 1. Changes in the analysis between versions are given in Figure 2. As seen in Figure 2, the technical debt in the nopCommerce application, which was initially measured as 5.54%, decreased to below 5% in version 4.30. However, with version 4.40, the technical debt ratio increased to 6.44% and continued to decrease with micro changes in subsequent versions. In version 4.80.0, it changed significantly and approached almost 7%. This trend shows that technical debt increases in major version transitions.

Table 1. Version-technical debt values for nopCommerce

Version	Debt
3.80	5,54%
3.90	5,67%
4.00	5,66%
4.10	5,69%
4.30	4,83%
4.40	6,44%
4.40.1	6,45%
4.40.2	6,45%
4.40.3	6,44%
4.40.4	6,45%
4.50.0	6,35%
4.50.1	6,35%
4.50.2	6,36%
4.50.3	6,35%
4.50.4	6,35%
4.60.0	6,34%
4.60.1	6,34%
4.60.2	6,34%
4.60.3	6,33%
4.60.4	6,33%
4.60.5	6,34%

ISSN: 2618-6055 / 9, (1), 25 – 40, 2025
DOI:10.62301/usmtd.1696804

4.60.6	6,34%
4.70.0	6,30%
4.70.1	6,30%
4.70.2	6,30%
4.70.3	6,30%
4.70.4	6,30%
4.70.5	6,33%
4.80.0	6,71%
4.80.1	6,70%

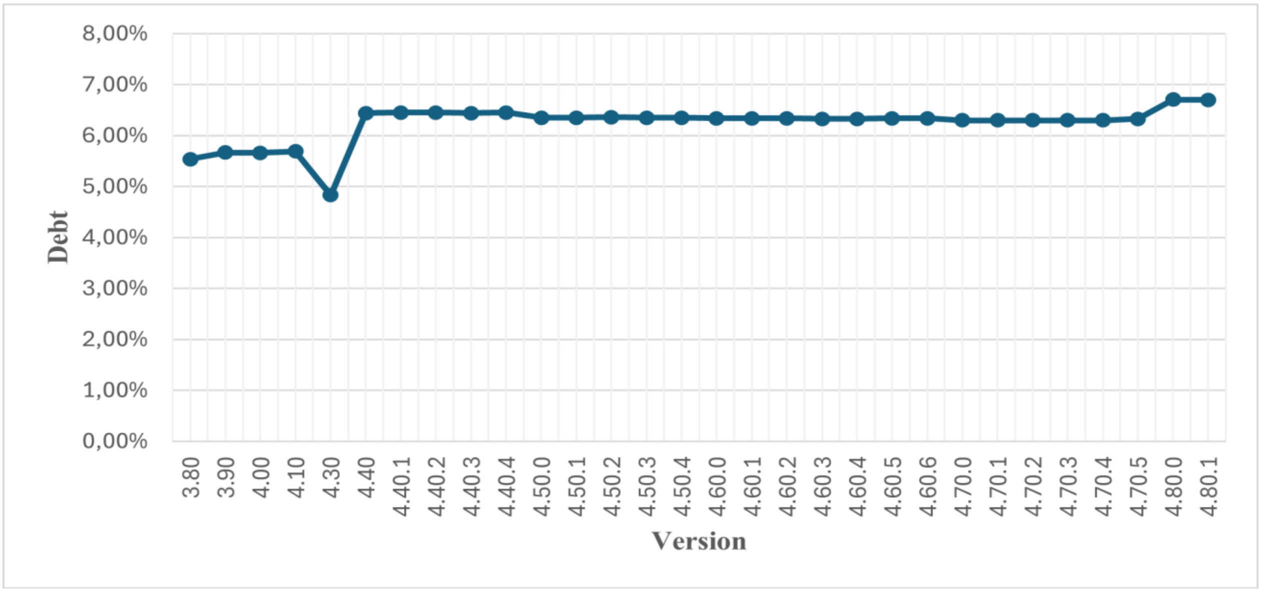

Figure 2. Version-based technical debt values of the nopCommerce application

The technical debt value for each version of OrchardCore is given in Table 2. Changes in the analysis between versions are given in Figure 3. As seen in Figure 3, the technical debt ratio in the OrchardCore application remained stable at 3.5% for a long time. Although it fell below 3.5% starting from version 1.6.0, it approached 3.5% again in version 1.8.0. A significant improvement in technical debt was seen in the transition to version 2.0.0, and the ratio fell to 2.9%. Since this version, the technical debt ratio has remained low with minor fluctuations.

Table 2. Version-technical debt values for OrchardCore

Version	Debt
0.0.1	3,53%
0.0.2	3,53%
0.0.3	3,53%
0.0.4	3,53%
1.0.0	3,53%
1.1.0	3,53%
1.2.0	3,53%
1.2.1	3,53%
1.2.2	3,53%
1.3.0	3,53%
1.4.0	3,58%
1.5.0	3,54%
1.6.0	3,47%
1.7.0	3,38%
1.7.1	3,38%
1.7.2	3,38%
1.8.0	3,45%
1.8.1	3,45%
1.8.2	3,46%

ISSN: 2618-6055 / 9, (1), 25 – 40, 2025
DOI:10.62301/usmtd.1696804

1.8.3	3,46%
1.8.4	3,43%
2.0.0	2,92%
2.0.1	2,91%
2.0.2	2,92%
2.1.0	3,08%
2.1.1	3,09%
2.1.2	3,09%
2.1.3	3,09%

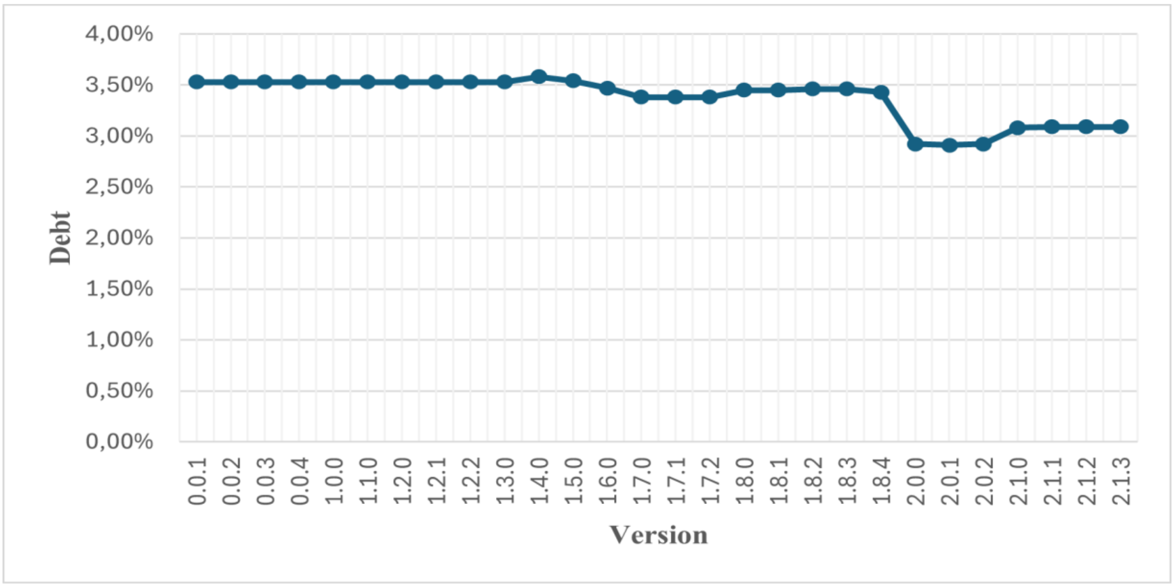

Figure 3. Version-based technical debt values of the OrchardCore application

The technical debt value for each version for RavenDB is given in Table 3. The changes in the analysis between versions are given in Figure 4. As can be seen in Figure 4, the technical debt in the RavenDB implementation was initially measured at 5.6% and increased to 5.84% in version 5.4.104. The technical debt ratio remained constant at this level in all subsequent versions and increased to 5.88% in version 5.4.202 with only minor fluctuations. This indicates a steady but limited improvement in debt management.

Table 3. Version-technical debt values for RavenDB

Version	Debt
5.4.100	5,60%
5.4.101	5,60%
5.4.102	5,61%
5.4.104	5,84%
5.4.105	5,84%
5.4.109	5,84%
5.4.110	5,85%
5.4.111	5,85%
5.4.112	5,85%
5.4.113	5,85%
5.4.114	5,85%
5.4.115	5,85%
5.4.116	5,86%
5.4.117	5,86%
5.4.118	5,85%
5.4.119	5,85%
5.4.202	5,88%
5.4.203	5,88%
5.4.204	5,89%

5.4.205	5,89%

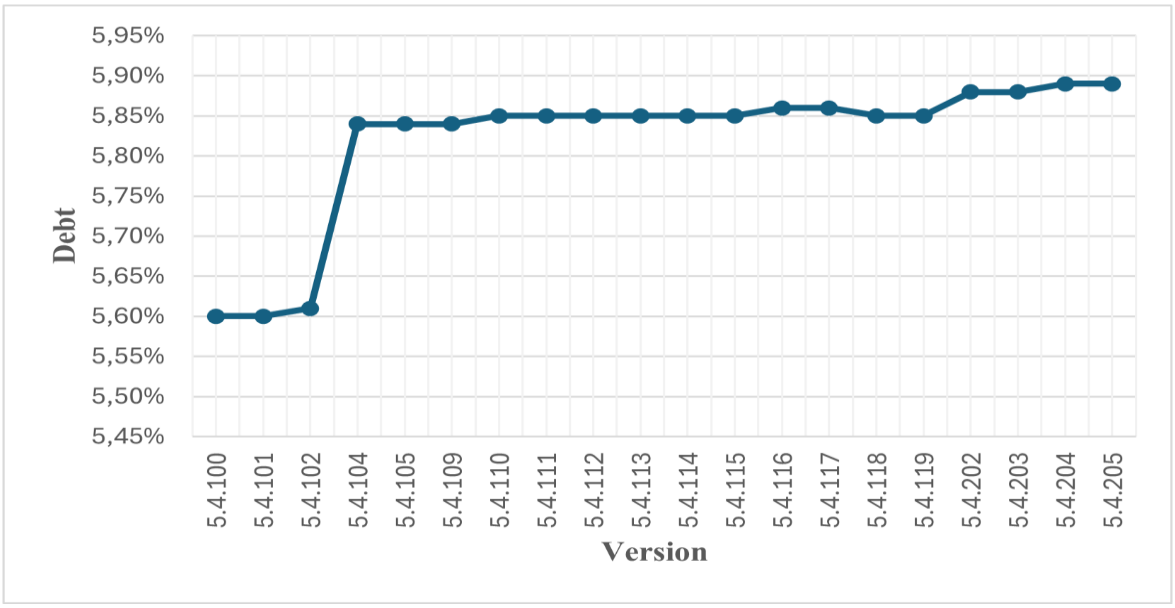

Figure 4. Version-based technical debt values of the RavenDB application

The technical debt value for each version for ShareX is given in Table 4. The changes in the analysis between versions are given in Figure 5. As seen in Figure 5, the technical debt in the ShareX application started at 5.77% in version 13.0.0 and decreased to 5.33% by version 13.2.0. It stabilized at 5.4% in subsequent versions. This trend shows that technical debt has been reduced and managed over time.

Table 4. Version-technical debt values for ShareX

Version	Debt
13.0.0	5,77%
13.0.1	5,77%
13.1.0	5,71%
13.2.0	5,33%
13.2.1	5,33%
13.3.0	5,37%
13.4.0	5,37%
13.5.0	5,38%
13.6.0	5,40%
13.6.1	5,40%
13.7.0	5,40%
14.0.0	5,47%
14.0.1	5,47%
14.1.0	5,46%
15.0.0	5,42%

ISSN: 2618-6055 / 9, (1), 25 – 40, 2025
DOI:10.62301/usmtd.1696804

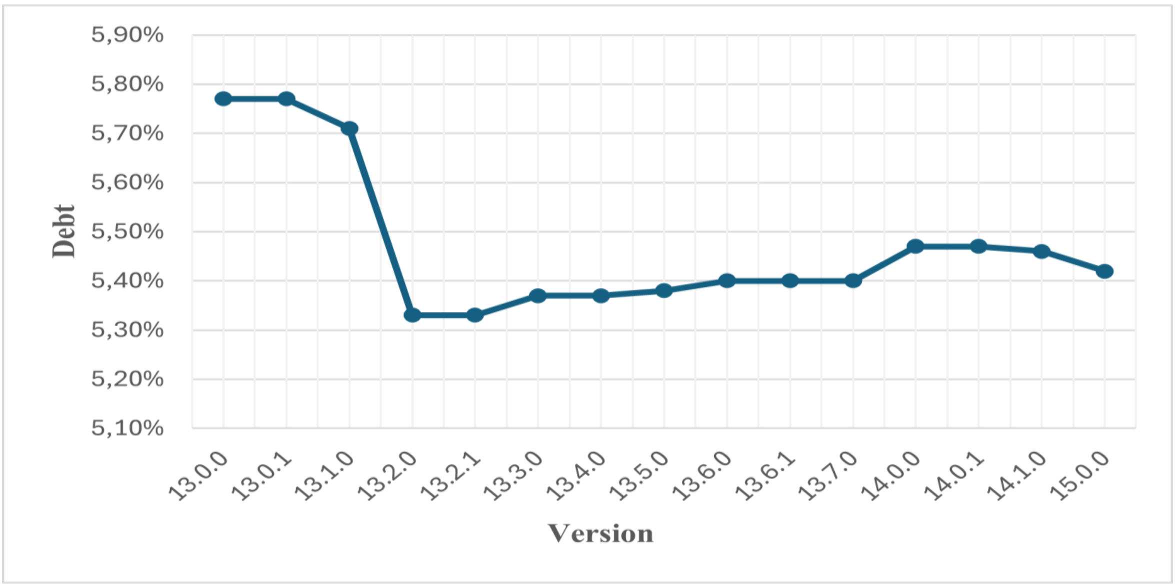

Figure 5. Version-based technical debt values of the ShareX application

The findings of this study reveal distinct technical debt trends across four open source .NET applications (nopCommerce, OrchardCore, RavenDB, and ShareX) analyzed across multiple versions using the NDepend static analysis tool. The analysis showed that technical debt does not follow a uniform pattern across all projects, but instead exhibits application-specific trends shaped by release strategy, architecture, and possibly contributor dynamics.

As seen in Figure 2, nopCommerce's technical debt percentage has shown significant increases in major releases. The debt level started at 5.54% in the first release and decreased to 4.83% in version 4.30, showing some initial improvements. However, there was a sharp increase to around 6.5% with the major release of 4.40, followed by slight decreases in minor releases. The second increase was observed in version 4.80, where the debt reached almost 7%. This pattern suggests that a significant amount of new debt is incurred during major release transitions, most likely due to significant feature additions or rapidly introduced architectural changes. In contrast, intervening minor releases allowed for incremental refactoring or stabilization that modestly reduced the debt. Overall, nopCommerce's trend has been upward over time, meaning that if proactive debt management is not implemented during major upgrades, the project accumulates "debt spikes" that increase the underlying debt level. These results highlight the need for careful planning during major releases. Large transitions often create complexity faster than it pays off, while smaller periods of change allow the team to pay off or stabilize some of its debt.

In Figure 3, OrchardCore has exhibited a remarkably stable and low technical debt profile. Over a long series of releases, the debt ratio has remained stable around 3.5%, which is significantly lower than other projects. There were minor fluctuations with a slight drop below 3.5% in version 1.6.0, then returning to 3.45% in version 1.8.0. In particular, a significant improvement occurred with the release of version 2.0.0. The technical debt ratio dropped to 2.92% and remained low in subsequent releases (with insignificant fluctuations in the range of 2.9–3.1%). This suggests that OrchardCore not only maintained a low debt level for a long time, but also took advantage of a major release to significantly reduce its technical debt. One possible reason is OrchardCore's highly modular architecture and emphasis on code quality, which can limit the spread of debt by isolating components. The modular design appears to be effective in debt management, as seen in the consistently low debt percentage. Moreover, the drop in v2.0.0 suggests that developers are using this milestone for significant refactoring or architectural optimization, fixing delays, and "paying off" debt. In summary, OrchardCore's case demonstrates that when solid architectural practices are in place, a project's technical debt can remain both low and stable, and major releases can serve as an opportunity to proactively reduce debt rather than incur it.

According to Figure 4, RavenDB's technical debt has remained relatively constant across the versions examined, with a slight upward drift. The debt percentage initially rose from 5.6% to 5.84% in version 5.4.104. Later, over the course of many incremental updates, the debt ratio remained essentially constant at this level with only minor fluctuations, gradually increasing to approximately 5.90% in the last

ISSN: 2618-6055 / 9, (1), 25 – 40, 2025
DOI:10.62301/usmtd.1696804

analyzed version. This trend suggests that RavenDB's maintainers have managed to avoid any dramatic accumulation of technical debt over time. The largely flat line suggests that as new features or changes are introduced, efforts are made to offset the new debt (e.g., through code reviews or minor refactorings) so that the overall debt remains constant. In other words, the team was likely paying the interest on the technical debt as it accrued, keeping the underlying debt amount under control. The slight increase from 5.6% to 5.89% indicates a modest accumulation. This could be due to the inherent complexity of a database system or a focus on performance optimizations rather than code cleanup. However, the lack of major increases suggests a disciplined development approach where technical debt is constantly monitored and managed with a slow burn. RavenDB's profile can be viewed as a stable state of technical debt. The project has not significantly reduced its current debt or allowed it to increase, reflecting a containment strategy.

Figure 5 shows that ShareX follows the exact opposite trajectory to nopCommerce. Its technical debt has decreased over time. Starting at 5.77% in version 13.0.0, the debt percentage has decreased to 5.33% in version 13.2.0. It has stabilized at 5.4% in subsequent versions and has remained consistently lower than its initial value, albeit with very small increases between versions. This downward trend suggests that ShareX developers actively improved code quality in early 13.x versions and paid off the debt. After version 13.0.0, the team seems to have made efforts to refactor and clean up the code in minor versions, thus eliminating code smells or inefficiencies and resulting in a measurable decrease in debt. Once this low debt base was reached, the project maintained it with only minor increases, suggesting that the new debt introduced in subsequent versions was roughly offset by ongoing maintenance. ShareX's trend exemplifies successful technical debt management during evolution. Contributing factors to this situation may include a relatively small scope, active community contributions focused on quality, or a release policy that values resolving technical debt issues over adding new features. The basic idea is that technical debt does not have to inevitably increase. Technical debt can be reduced and kept under control throughout a project's lifecycle with conscious effort.

In general, debt trends vary according to the type of software and development strategies. While debt can increase rapidly in frequently changing commercial applications such as NopCommerce, debt can remain at low levels in modular and planned projects such as OrchardCore. The RavenDB example shows that debt can progress in a controlled manner, while the ShareX example shows that debt can be reduced. These findings reveal that technical debt follows a different path in each project over time. The changes observed in each project are affected not only by the code itself, but also by the development approach, release frequency, and architectural preferences. Technical debt does not always have to increase; it can be reduced or kept under control with the right strategies.

Researchers have called for moving beyond metaphor to systematic management of technical debt. Kruchten et al. [19] emphasized the importance of technical debt in software engineering and that it should be treated as a fundamental part of project management rather than an abstract concept. Our multi-version analysis supports their claim. We find that unmanaged debt can undermine software quality, but systematically addressing debt (as done in OrchardCore or ShareX) yields tangible quality benefits. Our results highlight the importance of making technical debt visible and measurable in a practical way.

This study is also consistent with the literature investigating how to define and measure technical debt. Alves et al. [20] conducted a systematic mapping study of technical debt management and highlighted the need for indicators to measure and monitor debt in projects. This study addressed this need by using a static analysis tool (NDepend) to continuously measure debt across releases. This allowed for the identification of when and where debt changed, which is exactly the type of monitoring they advocated. This study also aligns with the empirical study of Ernst et al. [21] who examined practitioners' approaches to technical debt and highlighted that many teams struggle with whether to measure or manage it. Importantly, they found that static analysis tools can effectively uncover technical debt items and provide valuable support to developers. Our experience with NDepend confirms this; the tool was effective in uncovering hidden issues and trends that were difficult to track manually. The fact that the ShareX and OrchardCore teams were able to reduce debt suggests that they were likely paying attention to such tool feedback or similar quality signals, while nopCommerce's increases may indicate periods when such feedback was underestimated or overridden by feature pressure.

ISSN: 2618-6055 / 9, (1), 25 – 40, 2025
DOI:10.62301/usmtd.1696804

Another important issue is how the tools represent technical debt. Fontana et al. [22] discussed technical debt indices provided by tools and stated that these indices provide an aggregate view of a project's debt and can be calculated by various static analysis platforms. The technical debt percentage from NDepend used in this study is exactly such an index and is a composite measure that condenses a large number of code issues into a single debt percentage. The fact that we use this index across multiple releases demonstrates its practical utility. As suggested by Fontana et al., an index allows for high-level tracking of debt evolution and cross-comparison across projects or releases. By presenting real-world data that explain these concepts, this study strengthens the bridge between theoretical discussions of technical debt and the practical realities observed in open source projects.

4. Conclusion

This study examined the evolution of technical debt across multiple versions of four open-source .NET-based applications using version-based static analysis with the NDepend tool. The results revealed that each project exhibited distinct patterns of technical debt change, shaped not only by the version structure but also by project characteristics and potential architectural decisions. Rather than assuming that technical debt naturally increases over time, the findings show that ShareX and OrchardCore applications achieve debt reduction or stability throughout their evolution, indicating that effective debt control is feasible even in open-source, community-driven environments.

This section discusses what the increases and decreases in technical debt observed in previous sections of the applications mean. NopCommerce's noticeable increase in technical debt after major version migrations suggests that rapid feature delivery may be prioritized over code quality, likely due to limited time for refactoring or internal review. OrchardCore, on the other hand, showed a consistent and well-managed debt profile with a significant decrease around version 2.0. This suggests that the development team made a deliberate effort to improve the internal code structure through extensive cleanup or architectural improvements. RavenDB followed a more stable trajectory where small increases in debt gradually accumulated but no major spikes were observed. This suggests that the team was continuously implementing small improvements to prevent significant increases in technical debt and a strategy of gradual containment rather than aggressive reduction. ShareX presented the opposite case where debt levels gradually decreased over time, likely reflecting successful maintenance practices and attention to code quality between feature additions. These differences suggest that technical debt is not simply a natural consequence of software age or codebase size. Instead, it evolves as a result of project-specific development strategies, architectural decisions, and how proactively a team manages quality during release changes. Projects that incorporate regular refactoring or maintenance windows into their release cycles are in a better position to control or reduce technical debt.

In practical terms, these situations suggest that development teams should integrate technical debt tracking into their release planning, not as a general quality check, but with tools that can continuously monitor metrics across multiple dimensions. It should be noted that in modular platforms that undergo frequent integrations, such as CMS or e-commerce systems, technical debt is more likely to accumulate and should be taken into account during major release transitions. Furthermore, prioritization strategies, such as refactoring before release, should be clearly defined and integrated into the lifecycle. The findings of this study suggest that failure to do so risks long-term quality erosion that may not be apparent in short-term speed gains.

The changes in technical debt levels across versions in the analyzed applications are not only general trends, but also significant numerical differences. For example, in nopCommerce, the technical debt ratio decreased to 4.83% in version 4.30, but increased to 7% in version 4.80.0, approximately 2%, indicating that major version transitions trigger technical debt accumulation. In the OrchardCore application, technical debt remained at 3.5% for a long time, decreasing to 2.92% in version 2.0.0, showing an improvement of nearly 20%. In RavenDB, the debt ratio increased from 5.60% in the first version to 5.84% in version 5.4.104, and to 5.89% in the last version, showing a steady but limited increase of 0.3%. ShareX, on the other hand, achieved an absolute improvement of 0.44%, decreasing the rate from 5.77% in version 13.0.0 to 5.33% in version 13.2.0. These numerical findings reveal the effects of software architecture, version migration strategies, and maintenance processes on technical debt, and indicate that development teams should not only monitor trends but also analyze absolute changes.

As a result, it has been shown that technical debt is inevitable in software development processes but can be managed with the right strategies. It is recommended that teams handle development activities together with technical debt management, consider debt risks during version transitions, and develop regular follow-up mechanisms to ensure sustainable quality.

In future studies, similar analyses can be performed on different software types and development approaches, different analysis tools can be compared and debt management strategies can be considered from a broader perspective, and different applications can be analyzed with various metrics. Such studies will provide significant contributions to increasing the sustainability of software quality and to understanding the effects of technical debt on the software life cycle in more depth.

5. Acknowledgements

This study is derived from the master's thesis of Rafet GÖZBAŞI, a student of Dokuz Eylül University.

6. References

[1] A. Capiluppi, J.F. Ramil, Studying the evolution of open source systems at different levels of granularity: Two case studies, in: Proceedings of the 7th International Workshop on Principles of Software Evolution, IEEE, 2004, pp. 113–118.

[2] W. Scacchi, Free/open source software development, in: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2007, pp. 459–468.

[3] K.J. Stol, B. Fitzgerald, Inner source—adopting open source development practices in organizations: a tutorial, IEEE Softw. 32 (4) (2014) 60–67.

[4] M. Beller, R. Bholanath, S. McIntosh, A. Zaidman, Analyzing the state of static analysis: a large-scale evaluation in open source software, in: Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016, pp. 470–481.

[5] N. Ayewah, W. Pugh, J.D. Morgenthaler, J. Penix, Y. Zhou, Using findbugs on production software, in: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, 2007, pp. 805–806.

[6] B.S. Basutakara, P.N. Jeyanthi, A review of static code analysis methods for detecting security flaws, J. Univ. Shanghai Sci. Technol. 23 (6) (2021) 647–653.

[7] E. Sultanow, A. Ullrich, S. Konopik, G. Vladova, Machine learning based static code analysis for software quality assurance, in: Proceedings of the 13th International Conference on Digital Information Management (ICDIM), 2018, pp. 156–161.

[8] J. Yeboah, S. Popoola, Uncovering user concerns and preferences in static analysis tools: a topic modeling approach, in: Proceedings of the 2nd International Conference on Artificial Intelligence, Blockchain, and Internet of Things (AIBThings), 2024, pp. 1–6.

[9] V. Lenarduzzi, F. Pecorelli, N. Saarimaki, S. Lujan, F. Palomba, A critical comparison on six static analysis tools: detection, agreement, and precision, J. Syst. Softw. 198 (2023) 111575.

[10] M. Nachtigall, L. Nguyen Quang Do, E. Bodden, Explaining static analysis—a perspective, in: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), 2019, pp. 29–32.

[11] P.C. Avgeriou, D. Taibi, A. Ampatzoglou, F.A. Fontana, T. Besker, A. Chatzigeorgiou, et al., An overview and comparison of technical debt measurement tools, IEEE Softw. 38 (3) (2020) 61–71.

[12] N.A. Ernst, S. Bellomo, I. Ozkaya, R.L. Nord, What to fix? Distinguishing between design and non-design rules in automated tools, in: Proceedings of the IEEE International Conference on Software Architecture (ICSA), IEEE, 2017, pp. 165–168.

[13] T. Coulin, M. Detante, W. Mouchère, F. Petrillo, Software architecture metrics: a literature review, arXiv preprint arXiv:1901.09050 (2019).

ISSN: 2618-6055 / 9, (1), 25 – 40, 2025
DOI:10.62301/usmtd.1696804

[14] M.K. Debbarma, S. Debbarma, N. Debbarma, K. Chakma, A. Jamatia, A review and analysis of software complexity metrics in structural testing, Int. J. Comput. Commun. Eng. 2 (2013) 129–133.

[15] J. Ludwig, S. Xu, F. Webber, Compiling static software metrics for reliability and maintainability from GitHub repositories, in: Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC), 2017, pp. 5–9.

[16] E.Y. Hernandez-Gonzalez, A.J. Sanchez-Garcia, M.K. Cortes-Verdin, J.C. Perez-Arriaga, Quality metrics in software design: a systematic review, in: Proceedings of the 7th International Conference in Software Engineering Research and Innovation (CONISOFT), 2019, pp. 80–86.

[17] A.S. Nuñez-Varela, H.G. Pérez-Gonzalez, F.E. Martínez-Perez, C. Soubervielle-Montalvo, Source code metrics: a systematic mapping study, J. Syst. Softw. 128 (2017) 164–197.

[18] W. Cunningham, The WyCash portfolio management system, ACM Sigplan OOPS Messenger 4 (2) (1992) 29–30.

[19] P. Kruchten, R.L. Nord, I. Ozkaya, Technical debt: from metaphor to theory and practice, IEEE Softw. 29 (6) (2012) 18–21.

[20] N.S. Alves, T.S. Mendes, M.G. De Mendonça, R.O. Spínola, F. Shull, C. Seaman, Identification and management of technical debt: a systematic mapping study, Inf. Softw. Technol. 70 (2016) 100–121.

[21] N.A. Ernst, S. Bellomo, I. Ozkaya, R.L. Nord, I. Gorton, Measure it? manage it? ignore it? software practitioners and technical debt, in: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 50–60.

[22] F.A. Fontana, R. Roveda, M. Zanoni, Technical debt indexes provided by tools: a preliminary discussion, in: Proceedings of the 8th IEEE International Workshop on Managing Technical Debt (MTD), IEEE, 2016, pp. 28–31.

[23] J. Holvitie, S.A. Licorish, R.O. Spínola, S. Hyrynsalmi, S.G. MacDonell, T.S. Mendes, V. Leppänen, 2Technol. 96 (2018) 141–160.

[24] P. Avgeriou, I. Ozkaya, A. Chatzigeorgiou, M. Ciolkowski, N.A. Ernst, R.J. Koontz, F. Shull, Technical debt management: the road ahead for successful software delivery, in: Proc. 2023 IEEE/ACM Int. Conf. Softw. Eng.: Future Softw. Eng. (ICSE-FoSE), IEEE, 2023, pp. 15–30.

[25] P. Kokol, Software quality: how much does it matter?, Electronics 11 (16) (2022) 2485.

[26] H.J. Kang, K.L. Aw, D. Lo, Detecting false alarms from automatic static analysis tools: how far are we?, in: Proc. 44th Int. Conf. Softw. Eng. (ICSE), 2022, pp. 698–709.

[27] A. Melo, R. Fagundes, V. Lenarduzzi, W.B. Santos, Identification and measurement of Requirements Technical Debt in software development: a systematic literature review, J. Syst. Softw. 194 (2022) 111483.

[28] H.J. Junior, G.H. Travassos, Consolidating a common perspective on technical debt and its management through a tertiary study, Inf. Softw. Technol. 149 (2022) 106964.

[29] D.C. Yadav, Y. Singh, A.K. Pandey, A. Kannagi, Computerized software quality evaluation with novel artificial intelligence approach, Proc. Eng. 6 (1) (2024) 363–372.

[30] T. Honglei, S. Wei, Z. Yanan, The research on software metrics and software complexity metrics, in: Proceedings of the International Forum on Computer Science-Technology and Applications, IEEE, 2009, pp. 131–136.

[31] S.B. Pandi, S.A. Binta, S. Kaushal, Artificial intelligence for technical debt management in software development, arXiv preprint arXiv:2306.10194 (2023).

[32] nopCommerce, nopCommerce release tags, https://github.com/nopSolutions/nopCommerce/tags, 2025 (accessed 01.02.25).

[33] Orchard Core, Orchard Core release tags, https://github.com/OrchardCMS/OrchardCore/tags, 2025 (accessed 08.02.25).

ISSN: 2618-6055 / 9, (1), 25 – 40, 2025
DOI:10.62301/usmtd.1696804

[34] RavenDB, RavenDB release tags, https://github.com/ravendb/ravendb/tags, 2025 (accessed 15.02.25).

[35] ShareX, ShareX release tags, https://github.com/ShareX/ShareX/tags, 2025 (accessed 22.02.25).

[36] A. Juneja, B. Sondhi, A. Sharma, Nopcommerce customization for improved functionality and user experience, 2023.

[37] Orchard Core, Orchard Core Documentation, https://docs.orchardcore.net, 2025 (accessed 27.05.25).

[38] P. Nepaliya, P. Gupta, Performance analysis of NoSQL databases, Int. J. Comput. Appl. 127 (12) (2015) 36–39.

[39] Wikipedia, ShareX, https://tr.wikipedia.org/wiki/ShareX, 2025 (accessed 27.05.25).

[40] M. Stanković, Komparativna analiza alata za statičku analizu koda u svrhu identifikacije i procene tehničkog duga u .NET projektima, Zb. Rad. Fak. Tech. Nauka Novi Sad 37 (8) (2022) 1337–1340.

[41] R.H. Pfeiffer, M. Lungu, Technical debt and maintainability: how do tools measure it?, arXiv preprint arXiv:2202.13464 (2022).

[42] J. Lefever, Y. Cai, H. Cervantes, R. Kazman, H. Fang, On the lack of consensus among technical debt detection tools, in: Proc. 2021 IEEE/ACM 43rd Int. Conf. Softw. Eng.: Softw. Eng. Pract. (ICSE-SEIP), IEEE, 2021, pp. 121–130.

[43] R. Shaukat, A. Shahoor, A. Urooj, Probing into code analysis tools: a comparison of C# supporting static code analyzers, in: Proc. 15th Int. Bhurban Conf. Appl. Sci. Technol. (IBCAST), IEEE, 2018, pp. 455–464.

[44] L. Pavlič, T. Hliš, M. Heričko, T. Beranič, The gap between the admitted and the measured technical debt: an empirical study, Appl. Sci. 12 (15) (2022) 7482.

[45] D. Saraiva, J.G. Neto, U. Kulesza, G. Freitas, R. Reboucas, R. Coelho, Technical debt tools: a systematic mapping study, in: Proc. Int. Conf. Enterp. Inf. Syst. (ICEIS), vol. 2, 2021, pp. 88–98.