Quantifying Technical Debt: An Empirical Study using Debt Detective

Sridhar Chimalakonda
IIT Tirupati
Tirupati, Andhra Pradesh, India

Rajrupa Chattaraj
IIT Tirupati
Tirupati, Andhra Pradesh, India

Ganesh Priyatham P
IIT Tirupati
Tirupati, Andhra Pradesh, India

Vishnu Vardhan P
IIT Tirupati
Tirupati, Andhra Pradesh, India

Sarthak Girotra
IIT Tirupati
Tirupati, Andhra Pradesh, India

Sirish Sekhar
IIT Tirupati
Tirupati, Andhra Pradesh, India

Surya Varman E
IIT Tirupati
Tirupati, Andhra Pradesh, India

ABSTRACT

This Tool is designed to provide a comprehensive analysis of technical debt in software projects by considering multiple metrics such as community activeness, code vulnerability and security, deprecated and outdated code, and code quality and standard. Technical debt is a pervasive issue in software development, where it occurs due to the accumulation of incomplete solutions, quick fixes, and shortcuts, leading to an increase in the cost of software development over time.The tool facilitates the measurement of technical debt by analyzing the above-mentioned metrics to quantify the extent of technical debt in a software project. This is achieved through the collection of data from open source software projects and databases, allowing for a robust and thorough analysis of the software project.By utilizing the Technical Debt Quantification Tool, developers can gain insights into the extent of technical debt present in their code-base, and prioritize their efforts to reduce technical debt by focusing on the areas that require the most attention. This enables developers to take proactive measures to address technical debt, resulting in improved software quality and reduced long-term development costs.

ACM Reference Format:
Sridhar Chimalakonda, Rajrupa Chattaraj, Ganesh Priyatham P, Vishnu Vardhan P, Sarthak Girotra, Sirish Sekhar, and Surya Varman E. 2018. Quantifying Technical Debt: An Empirical Study using Debt Detective. In . ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Technical debt refers to the accumulation of poor design choices and shortcuts in software development, which can reduce code quality and maintainability. These costs can be intentional or unintentional, and and its impact can be felt throughout the life cycle of a software project.

The impact of technical debt on a software project can be significant. In the short term, it can increase the time and effort needed to update and fix bugs in the code-base, resulting in increased delays and costs Technical cost delivery time length can reduce software quality, making software more difficult to maintain and develop over time. This can lead to higher costs and reduced competitiveness.

Furthermore, technical debt can also affect the morale of the development team, as they may become frustrated with the lack of complexity and clarity in the codebase This can increase turnover rates, making it harder to attract new talent that has come into the industry.

It is important to manage technical debt in advance, make conscious decisions about when and how to create them, as well as establish a long-term repayment strategy This can help with the impact it has on the business reduced overhead and ensure the long-term success of the software.

2 RELATED WORK

[4] investigates the technical debt associated with security weaknesses, our tool tries to pick some parameters to quantify the technical debt associated with security weaknesses. [1] A comprehensive analysis of the different tools available to assess code quality and presenting the people with a comparison of where each tool's strength lies.

Though all the present papers just provide a comparison between existing tools, none of them truly actively try to quantify actionable technical debt(debt that one can address), this is our attempt to try to present the user with available popular metrics for their specific needs and assess how it impacts their software development lifecycle. In this way, we hope to empower developers to take meaningful actions to address technical debt in their projects.

3 DESIGN AND DEVELOPMENT

Quantifying technical debt can be a challenging task, but selecting appropriate parameters can help provide insight into the impact of poor practices and neglected package migration. Our efforts were primarily focused on five metrics which include addressing issues related to packages, such as outdated dependencies and compatibility issues, and secondarily on addressing issues related to code, such as poor code quality, security, and complexity. By prioritizing both areas, we were able to ensure the code base was maintainable and

, ,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

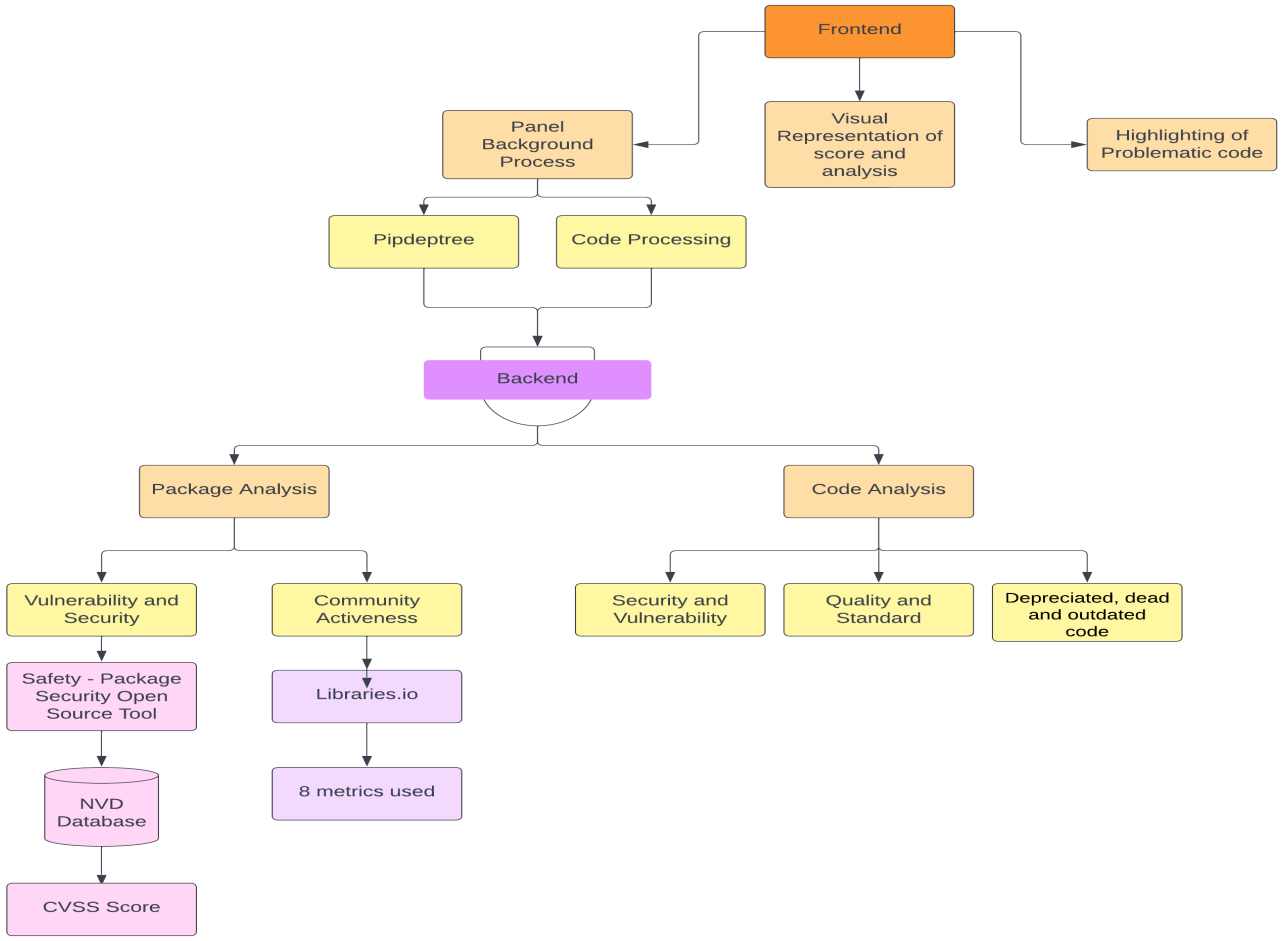

Figure 1: Development of Debt Detective

up-to-date. All the parameters have been listed below and the rationale behind choosing each one of them will be comprehensively discussed.

Package Parameters:

- Package Vulnerability and Security
- Community Activeness and Maintenance

Code Parameters:

- Depreciated, dead and outdated code
- Code vulnerability and security
- Code quality and standard

If left unchecked for an extended period of time, the packages that are installed and utilized throughout the software development life-cycle can have a substantial impact on the code base. Failing to update the code base with the latest versions of packages can increase the risk of security vulnerabilities remaining in the code, which can create opportunities for hackers to exploit the vulnerabilities and compromise the security of the system. Additionally, this can lead to data breaches, which can have serious consequences.

(1) **Package Vulnerability and Security**:
We utilized an existing tool called Safety to assess the security standard of a package. The tool called Pipdeptree was utilized to provide security with the package names and their corresponding versions. Safety gives us a well-detailed analysis of security vulnerabilities and their CVE-ID. This enables us to fetch the CVSS(Common Vulnerability Scoring System) score from the NVD(National Vulnerability Database) database based on the CVE-IDs of the vulnerabilities. Using this score, we analyze the impact of the package on the product development life-cycle and provide the user with a consolidated advisory. This approach helps to present a comprehensive and clear understanding of the package's impact on the development process in terms of security.

(2) **Community Activeness and Maintenance**:
The impact of community activeness and maintenance of a package on technical debt is a crucial aspect of software development. A package with an active and engaged community of developers and maintainers tends to have a lower

Exploitability Score	• Availability Impact
	• Integrity Impact
	• Confidentiality Impact
Impact Score	The potential impact of a security vulnerability on the target system

Figure 2: Sub-parameters for Package Vulnerability and Security

technical debt compared to packages with inactive communities. An active community can contribute to the package's documentation, and provide updates, bug fixes, and security patches, which can help reduce technical debt. Thus, the community's activeness and the maintenance of a package are essential factors to consider when evaluating the potential technical debt associated with using a package in software development.

- **Contributors:** Technical debt can increase when there is a lack of contributors or when contributors are not actively improving the code base. Having more contributors actively working on a code base can help reduce technical debt by bringing in fresh perspectives and ideas, distributing the workload, and improving code quality through code reviews and testing.
- **Recent Releases:**Technical debt can accumulate when a code-base is not regularly updated or maintained. A lack of recent releases may indicate that the project is not being actively maintained, and therefore, may have accumulated technical debt over time.
- **Stars and Forks:** A project with a large number of stars and forks often means that it has been widely adopted by developers who have tested the code-base and found it to be stable and of high quality. Additionally, a large user base can lead to more feedback, bug reports, and suggestions for improvement, which can help to identify and address technical debt more quickly.
- **Read-me:**A well-written read-me file can help reduce technical debt by providing clear instructions for contributing, testing, and maintaining the code base.
- **Age of Repository:** The age of a repository can provide useful insights into the stability and maturity of a code base. An older repository may suggest that the code base has been tested and refined over time, leading to a more stable and reliable system. Additionally, a well-maintained older repository with regular updates and bug fixes can indicate that the code base is actively being monitored and improved, which can help to reduce technical debt.
- **Multiple versions:** Having multiple versions of a project can be an indicator that the code base is being constantly updated so that new versions of the project are being released and maintained so that the previous bugs have been thoroughly checked and fixed.
- **Dependencies:** Technical debt can also accumulate when there are outdated or unmaintained dependencies. It's important to regularly update dependencies to ensure that they are secure and compatible with the code base. Additionally, having a large number of dependencies can also lead to technical debt, as it can be difficult to manage and maintain all of them.

Coding practices can have a significant impact on technical debt. Technical debt can arise due to bad coding practices, over-complex technical design et cetera. Our tool analyses the technical debt contributed by the code using the following parameters -

(1) **Dead and outdated code**: We use an existing python library Vulture which finds unused code in Python programs. Dead and outdated code can affect a software project in the following ways -
   - **Maintenance Costs:** Outdated and dead code can become difficult and expensive to maintain over time. As technology advances and new software frameworks are introduced, outdated code may need to be updated or rewritten to keep up with the latest standards. Dead code, on the other hand, can accumulate in a codebase over time and become a source of confusion for developers who may not know if the code is still being used or not. This can lead to unnecessary time and effort spent trying to understand and debug the code, which adds to technical debt.
   - **Security Risks:** Outdated and dead code can also pose security risks. If the code relies on outdated libraries or frameworks that have known vulnerabilities, it can leave the system open to attack. Similarly, dead code that is not being actively maintained can become a breeding ground for bugs and vulnerabilities that can be exploited by attackers.
   - **Reduced agility:** Outdated and dead code can make it harder to make changes to a system. When developers are working with a codebase that contains outdated or dead code, they may need to spend more time figuring out how the code works and what changes need to be made. This can slow down development and reduce the agility of the team, which can add to the technical debt over time.

(2) **Code vulnerability and security:** We used an existing tool Bandit, designed to find common security issues in Python

Age of repository	0/1 (at least 6 months)
Contributors	log2(number of contributors)
Dependents	2*log2(dependent_packages)
Multiple versions	0/1
Readme	0/1
Recent Release	0/1
Stars and Forks	log2(stars)+log2(forks)

Figure 3: Sub-parameters for Community Activeness and Maintenance

code. Code vulnerability and security can affect a software project in the following ways -

- **Time and resources spent on fixing security issues:** If security vulnerabilities are discovered in code, they need to be fixed as soon as possible to avoid potential data breaches or cyber-attacks. This takes time and resources away from other development tasks, creating technical debt. The cost of fixing security issues can be high, especially if the issues are discovered late in the development process.
- **Delayed deployment:** If code security issues are discovered during the testing phase or after deployment, it may delay the release of the code until the issues are fixed. This can create technical debt by delaying the delivery of new features or updates to customers or users.
- **Increased complexity:** Addressing security issues often requires adding additional code to the existing codebase, which can increase complexity. This additional complexity can make it harder to maintain the code over time, leading to more technical debt.

  Score Calculation: The security score calculated is done using two parameters, How severe the security/ vulnerability issue is and how confident of the issue we are. We calculate the severity score based on how severe the issue is (LOW: 1, MEDIUM: 3, and HIGH: 5), and similarly, we calculate the confidence score for each of the security issues. Using this we multiply the scores for each issue, which is the security score for one security issue, similar to this we calculate for all the issues present and find the fraction and get the score out of 100.

- **Code quality and standard**: We do a static code analysis of the project to determine the code quality, look for code smells and make suggestions about how the code could be refactored. We do this with the help of Python library pylint. Code quality and standards can affect a software project in the following way -
  - **Increased time and effort for maintenance:** Code that is poorly written, hard to understand, or lacks proper documentation can take longer to maintain and modify. This can create technical debt by increasing the time and effort required to fix bugs, add new features, or make changes to the code base.
  - **Higher likelihood of bugs and errors:** Bad code quality can also lead to an increased likelihood of bugs and errors, which can result in technical debt in the form of additional development time and costs to fix them.
  - **Difficulty in testing:** Poorly written code can be difficult to test, which can lead to additional technical debt in the form of increased time and effort required to ensure that the code works as intended.

4 USER SCENARIO

The Debt Detective tool is a VS Code extension that seamlessly integrates into a user's development environment. One of the most useful features of Debt Detective is its analysis panel, which is conveniently located on the left-hand side of the screen. It is a comprehensive source of information about the code being reviewed. It provides a detailed overview of various aspects related to code quality, security, and community engagement. The user is presented with a new visual representation of technical debt through a graph called code entropy graph which is essentially a radar chart whose area is a measure of the current technical debt associated with the code. To help users quickly identify problematic code, Debt Detective employs a highlighting system that draws attention to potential issues. This feature helps developers quickly identify and address potential problems in their code, making it easier to write clean, high-quality software. Debt

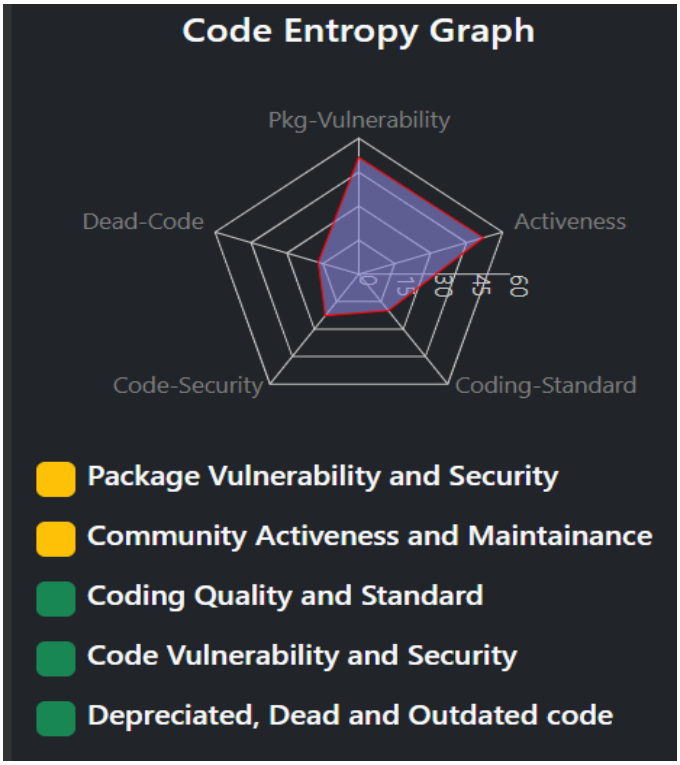

Figure 4: Code Entropy Graph

ISSUE SEVERITY/CONFIDENCE	SCORE
LOW	1
MEDIUM	3
HIGH	5

Figure 5: Code Security Weightage

Detective also offers a virtual environment creation feature, which allows users to create virtual environments with a simple keyboard command. Unlike the built-in feature of Vs-code our tool gives more control and information to the user on the latest stable version of the packages and whether they want to migrate to this version or not. This feature makes it easy for developers to work with different versions of libraries and packages, helping to prevent compatibility issues and other problems that can arise when working with complex software systems. To use Debt Detective, simply open the file you want to analyze and press the tool's icon. Debt Detective will automatically begin analyzing the code and highlighting potential issues in your code thus ensuring your code is clean, efficient, and of the highest quality.

5 DISCUSSION AND LIMITATIONS

In conclusion, Debt Detective is a powerful tool that can greatly benefit developers by helping them to identify potential issues in their code and improve its overall quality. While attempting to provide a thorough analysis of potential contributors to technical debt, the tool is limited in its ability to identify all types of technical debt. It is designed to detect specific types such as code smells and security violations, but it may overlook other forms of technical debt. Additionally, the tool may not consider the unique context of the software system, including its domain, architecture, and business objectives.

6 CONCLUSION AND FUTURE WORK

Technical debt is a significant problem in software development that can have serious negative impacts on software

quality, maintenance costs, and overall project success. It is therefore necessary to keep a track of technical debt and repay it for the success and sustainability of a project.
A potential improvement that significantly improves the tool's capabilities is by introducing a feature that will allow the code to be interpreted by an AI model to allow suggestions that could automatically give the best possible fix for a certain problem.

REFERENCES

[1] Paris C Avgeriou, Davide Taibi, Apostolos Ampatzoglou, Francesca Arcelli Fontana, Terese Besker, Alexander Chatzigeorgiou, Valentina Lenarduzzi, Antonio Martini, Athanasia Moschou, Ilaria Pigazzini, et al. An overview and comparison of technical debt measurement tools. *Ieee software*, 38(3):61–71, 2020.

[2] Kowndinya Boyalakuntla, Meiyappan Nagappan, Sridhar Chimalakonda, and Nuthan Munaiah. Repoquester: A tool towards evaluating github projects. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 509–513. IEEE, 2022.

[3] Bill Curtis, Jay Sappidi, and Alexandra Szynkarski. Estimating the size, cost, and types of technical debt. In *2012 Third International Workshop on Managing Technical Debt (MTD)*, pages 49–53. IEEE, 2012.

[4] Clemente Izurieta, David Rice, Kali Kimball, and Tessa Valentien. A position study to investigate technical debt associated with security weaknesses. In *Proceedings of the 2018 International Conference on technical debt*, pages 138–142, 2018.

[5] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *Ieee software*, 29(6):18–21, 2012.

[6] Miltiadis Siavvas, Dimitrios Tsoukalas, Marija Jankovic, Dionysios Kehagias, Alexander Chatzigeorgiou, Dimitrios Tzovaras, Nenad Anicic, and Erol Gelenbe. An empirical evaluation of the relationship between technical debt and software security. In *9th International Conference on Information society and technology (ICIST)*, volume 2019, 2019.

[7] Anders Sundelin, Javier Gonzalez-Huerta, and Krzysztof Wnuk. The hidden cost of backward compatibility: when deprecation turns into technical debt-an experience report. In *Proceedings of the 3rd International Conference on Technical Debt*, pages 67–76, 2020.