UNIVERSITY OF MACEDONIA
SCHOOL OF INFORMATION SCIENCES
DEPARTMENT OF APPLIED INFORMATICS

Evaluation of Code Quality and Hotspot Prioritization using repository mining

Bachelor's thesis

of

Archontis Emmanouil Kostis

Thessaloniki, February 2024

Evaluation of Code Quality and Hotspot Prioritization using repository mining

Archontis Emmanouil Kostis

Undergraduate Student of Applied Informatics at University of Macedonia

Bachelor’s Thesis
submitted for the partial fulfillment of its requirements

BACHELOR'S DEGREE IN APPLIED INFORMATICS

Supervisor
Alexander Chatzigeorgiou

Approved by the three-member examination committee on 14/02/2024

Alexander Chatzigeorgiou	Apostolos Ampatzoglou	Stylianos Xinogalos
....	....	....

Archontis Emmanouil Kostis

....

Abstract

Software development is characterized by continuous changes and improvements to meet evolving requirements and address emerging issues. Software repositories contain historical and valuable information about the overall development of software systems. To proactively manage code quality and address potential challenges, this thesis presents a service-based tool that employs repository mining techniques, using the Python Framework PyDriller, to assess code quality and prioritize hotspots in GitHub repositories. This thesis proposes a tool, named CodeInspector, the proposed tool aims to assist software engineers and researchers in identifying critical areas in their codebases that require immediate attention and understand the impact of individual commits on the overall maintainability of the software system. We achieve this by analyzing complexity and churn metrics and employing the Delta Maintainability Model (DMM) to assess code changes. The thesis emphasizes the importance of code quality in software development, equipping developers and researchers with a powerful tool that empowers them to create and maintain high-quality software systems.

Keywords: Python, Java, code quality, static code analysis, multi-metric analysis, PyDriller, Mining Software Repositories, GitHub, service tool, software development, software quality assurance

Preface

At the end of this remarkable and fascinating journey, I am both humbled and thrilled to present this thesis, the culmination of hours of dedication, research, and effort. This thesis would not have been possible without the support, encouragement, and aid of many people who have stood by me throughout this journey.

First and foremost, I would like to express my profound gratitude to Mr. Alexander Hatzigeorgiou, whose leadership and mentorship have served as the foundation of this thesis. His knowledge and assistance have been invaluable in crafting my work, and I am grateful for his support. Furthermore, I would like to convey my gratitude to two outstanding individuals, Nikolaos Nikolaidis Theodoros Maikantis, both Software Engineering Researchers at the University of Macedonia. Their views, and collaborative attitude is indispensable.

My friends and colleagues, George David Apostolidis and George Fakidis, have been pillars of strength and support throughout my journey. To my dear friend, Helen Vagiannopoulou, whose constant emotional and psychological support has been a guiding light, I am profoundly grateful. My heartfelt thanks go to my parents for their continuous support and numerous sacrifices, without which none of this would have been possible.

Last but not least, I'd like to express my gratitude to all my friends, whose companionship and encouragement have been a source of motivation and inspiration for me. This thesis is a monument to the power of teamwork, mentorship, and friendship. It is my goal that the knowledge and thoughts included within these pages will help to improve both the field and the community as a whole.

Table of contents

List of Figures

List of Code Blocks

List of tables

1 Introduction

Software development is a dynamic field that adapts to the changing needs and requirements of the digital world. This fast-paced industry is characterized by a never-ending cycle of modifications, updates, and enhancements aimed at producing higher-performing, more efficient, and feature-rich software systems. Understanding and maintaining code quality has proven critical in assuring the long-term survival and success of software projects. As a result, The need for effective tools to evaluate and prioritize code quality has expanded dramatically in today's world, where sophisticated, large-scale software systems are becoming the rule rather than the exception. As systems become more complex, maintaining high-quality code becomes increasingly difficult. The repercussions of inadequate code quality can be extensive both technically and economically, impacting not only the project's functionality but also its ability to adapt and endure in the face of future changes and challenges. High-quality code serves as the foundation for successful software, assuring durability, ease of maintenance, and efficient resource management. As a result, software engineers and project managers must be able to identify and prioritize portions of the codebase that require change or immediate attention.

Moreover, software repositories are massive information storage facilities, including the entire history of a project's evolution, from its genesis to its current iteration. This historical repository contains a wealth of data, providing a full view of the amazing journey of a software system's development history. This plethora of historical material is a veritable goldmine for researchers and engineers attempting to decipher the perplexing complexities.

1.1 Background and Motivation

GitHub[1], a well-known web-based hosting service based on the Git[2] version control system, is a key platform in open-source development and collaborative software engineering. The importance of GitHub in the modern software development landscape cannot be overstated. GitHub, holds an impressive repository count exceeding 300 million,[3] and has emerged as one of the most important sources of software artifacts on the Internet. The platform has permanently changed the nature of how development teams cooperate, share, and manage their codebase. As a result, GitHub has taken a significant position in both open-source and collaborative software development, leaving an effect on how projects are managed and maintained. The enormous repository ecosystem provided by GitHub gives up a world of possibilities for software engineering research.

Each repository is filled with historical data, including commit revisions, code reviews, conversations, and issue tracking. This plethora of data acts as a valuable source of knowledge, allowing researchers and engineers to obtain remarkable insights into the complicated evolution and lifecycle of software repositories. The deposit of code modifications, version histories, and collaborative development data offers a unique look at how software projects evolve over time. Mining this treasure trove of data can provide software experts with insights that allow them to make decisions about quality enhancements and task prioritization. This historical perspective, which spans a project's full existence, provides developers with the information required for strategic planning and the identification of crucial areas.

In software not all components are created equal, and some classes or files tend to be more problematic than others, leading to difficulties in maintenance, defects, and delays. Effective resource management and software quality enhancement are dependent on the ability to identify and prioritize these problem areas. Software quality management is critical to guaranteeing the long-term performance and maintainability of software projects and by proactively monitoring software quality, development teams can easily identify repeating patterns or target portions of their codebase that require improvement. This strategy eventually results in more profitable projects and the efficient utilization of resources. This paper presents an approach for prioritizing problematic classes and files within software repositories based on the ideas encapsulated in the Eisenhower matrix. The proposed method is based on code complexity and churn (how many times a file has changed on the version control system), and provides a structured and data-driven strategy for dealing with “hotspot files”. By incorporating this methodology into the development process, teams can use these insights to improve their overall code quality, strengthening the basis of software success in an ever-changing technological context.

1.2 Problem Statement

In today's software development landscape, the ever-increasing complexity and scale of modern software systems pose significant challenges for developers, project managers and software engineers. Manual code analysis, while necessary for quality assurance, is labor-intensive and time-consuming, making it impracticable for the massive codebases like in today’s software systems. Even the most careful human reviewer can be overwhelmed by the sheer volume of code included within these large-scale software systems. In recent years, automated code analysis tools have grown to be essential resources for developers. These tools are designed to quickly and reliably identify quality issues, expediting the problem-solving process. However, there is a considerable gap in how well many of these technologies utilize the full potential of repository mining and historical data. This can limit their ability to provide an in-depth understanding of code quality and also limit developers' ability to make well-informed decisions. This deficiency reduces the productivity of development processes and the development of strong and robust software, stressing the vital need for a more holistic data-driven approach.

In large software systems, some classes and files present more defects than others. These units, distinguished by their tendency of causing maintenance issues, errors, and delays, provide challenges throughout the whole development lifecycle. Effective management of these areas is critical for resource optimization and quality improvement. The identification of these units is only one step and prioritizing their resolution is equally important as the durability of software products is rooted in strategic prioritizing. Development teams can systematically improve the overall quality of their codebase by allocating focused attention and resources to these critical areas. The end result is not only more successful projects, but also more efficient and profitable ones.

1.3 Purpose – Objectives

The primary objective of this thesis is to develop a comprehensive and language-agnostic tool for code quality assessment and hotspot identification, utilizing mining techniques on software repositories. The proposed tool aims to provide software engineers and researchers with a solution for maintaining code quality, gaining important insights into code changes, and strategically prioritizing pivotal regions inside software projects. This multidimensional project aims to address the increasingly complicated and dynamic landscape of software development, where large-scale, collaborative projects are the norm and code quality is critical for long-term success.

Specifically, the primary objectives are:

1. **Develop a Flexible service-based Tool:**
   The primary goal is to create a versatile, service-based tool that can analyze code written in different programming languages. Despite the tool's primary focus on language-agnostic analysis, it will expand its support to programming languages compatible with PyDriller[4], assuring its applicability to a broad range of codebases.
2. **Extract Data from GitHub Repositories:**
   Another essential step is the extraction of comprehensive information from GitHub repositories. This covers a deep dive into commit history, churn, complexity, and Delta Maintainability Model (DMM) metrics[5]. This method will give the tool with the raw materials it needs to conduct a more in-depth code quality analysis.
3. **Implement a Hotspot Identification Mechanism:**
   The creation of a hotspot identification mechanism is an essential component of the proposed tool. Our approach will rely on complexity and churn metrics to identify critical files inside the codebase. This can help development teams to manage resources effectively and address the most crucial areas of the project first by prioritizing these areas.
4. **Leverage the Delta Maintainability Model (DMM)[5]:**
   Another important aspect of the thesis is the utilization of the Delta Maintainability Model (DMM)[5] in the commit analysis process. This model provides a thorough framework for measuring the impact of individual code modifications on a system’s maintainability. By incorporating DMM into the tool's functionality, it will be able to provide crucial insights into how code changes affect the project's general health and quality.

1.4 Contribution

The primary contribution of this thesis lies in the field of hotspot prioritization in software engineering. Our approach involves the creation of a tool capable of identifying and prioritizing problematic files within GitHub repositories, using complexity and churn metrics as guiding criteria. This methodology tries to identify classes or files contained inside the codebase that exhibit the simultaneous characteristics of high complexity and frequent modifications, indicating possible maintainability difficulties. By focusing on these critical areas, our solution helps to improve resource allocation and enhances the overall efficiency of software development processes. In essence, this method seeks to build the groundwork for the development of more maintainable and efficient software systems, assuring their lifetime and adaptability.

Complementing our holistic approach to improve code quality, we seamlessly integrate the Delta Maintainability Model (DMM) into the analysis process. This feature allows for a thorough assessment of individual code changes (commits) while taking into consideration DMM's comprehensive maintainability criteria. Each commit is evaluated using a rating system, with four separate levels: "EXCELLENT," "GOOD," "FAIR," and "BAD." This system provides developers with useful insights into the implication of each change, acting as a guide for decision-making during code reviews and influencing the direction of future development phases. The adoption of DMM strengthens the quality enhancement process, guaranteeing that maintainability remains at the center of software development activities, hence enriching the system's long-term success.

Finally, the thesis presents a holistic approach to hotspot prioritization and commit analysis. We hope to empower development teams to address difficult portions of their codebase proactively by using both static code analysis and repository mining data. Furthermore, the Delta Maintainability Model integration improves the analysis process by providing developers with a clear, metric-driven framework for reviewing individual changes in the codebase. This thesis provides a step forward in the construction of software systems that are not only efficient and adaptive, but also resilient in the face of changing requirements and technological breakthroughs. Finally, we hope to build a culture of data-driven decision-making and continual improvement in the ever-changing world of software engineering.

2 Bibliographic Review: Theoretical Background

2.1 Programming Languages

Programming languages are the fundamental building blocks of the software development industry. These languages have different features, syntax, and paradigms, each providing a unique set of tools for programmers to construct and design applications. They are the foundation of modern technology, allowing the development of a wide range of applications, from web and mobile apps to complex algorithms and systems.

Java, for example, is a general-purpose object-oriented programming language created in the 1990s by Sun Microsystems' James Gosling and others[6]. The technology was essential in revolutionizing the Internet and has greatly influenced the way software is developed. The language was created with portability in mind and facilitates application programming in a distributed computing environment. The goal was to create a computing environment that allows a program to be *written once, run anywhere*[7]. Platform independence is achieved by compiling Java code into Java bytecode, which is an optimized set of instructions that can be executed on a Java Virtual Machine.

2.1.1 Evolution and Impact

Programming languages are marked by a series of milestones and each language has played its role in redefining the boundaries of what can be achieved. Java, a general-purpose object-oriented programming language[6], pioneered concepts such as write once, run anywhere (WORA)[7]. This allows compiled code to transcend platform-specific limitations[8]. This paradigm-shifting technique greatly decreased implementation requirements, allowing developers to construct resilient and secure software solutions that can adapt to a variety of contexts. Programming languages' adaptability and benefits increase their appeal to a wide range of applications, ranging from the complexities of client-server web systems to the resilience of enterprise solutions and the precision of scientific computers.

2.1.2 Popularity in the Developer Community

Programming languages' importance extend beyond their technical aspects and are represented in their positions within the software developer community. Leading surveys, such as the *2023 Stack Overflow Developer Survey*[9] and Statista's *2023 Worldwide Developer Survey*[10], constantly highlight popular languages, demonstrating their prevalence among respondents. It is worth noting that JavaScript, Python, and Java are among the most popular languages, suggesting their importance and influence on the worldwide developer community. Additionally, according to Forbes[11], Java was the second most popular language in the world in February 2022, and is expected to continue growing in the future[12].

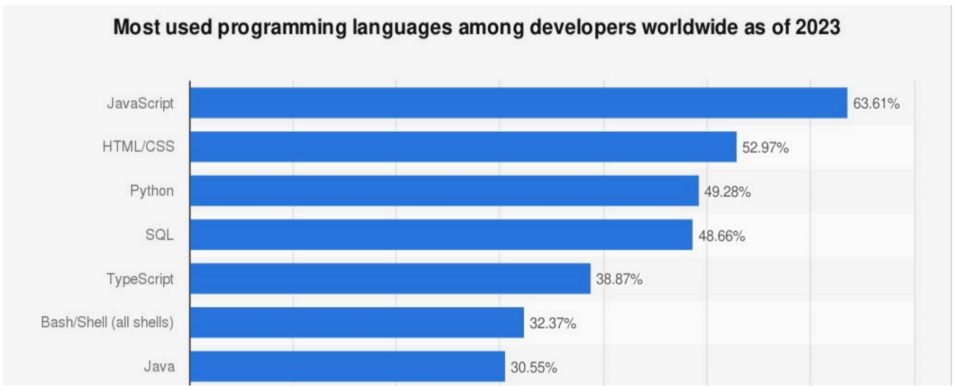


Figure 1 - Top Programming Languages (All Respondents) | Statista

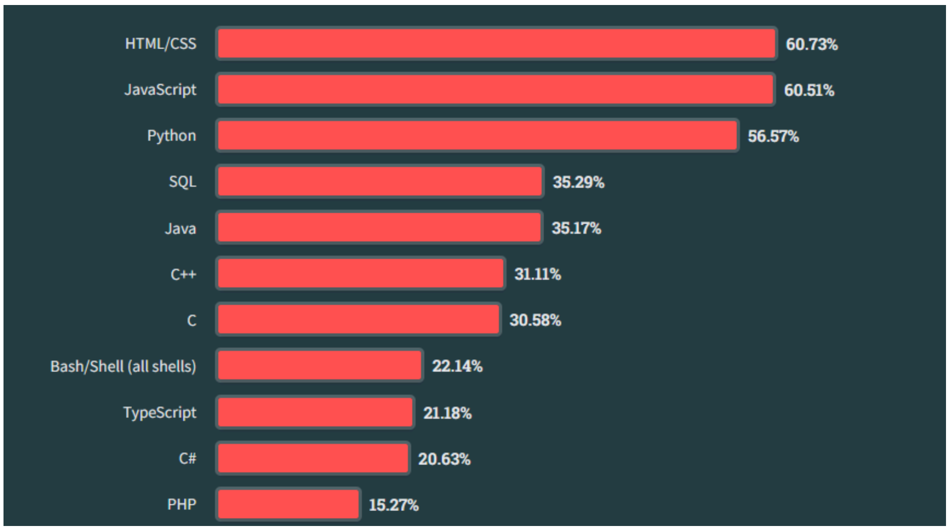


Figure 2 - Top Programming Languages (Users Learning to Code)

Another ranking system that measures the popularity of programming languages is the “*PYPL PopularitY of Programming Language Index”*[13]. PYPL is a system based on the number of searches for language tutorials on Google. According to the index Python, Java and Javascript remain the most popular languages among all. More specifically Java is the 2nd most popular programming language, in October 2023. During the years 2021 and 2022, the language has grown by 1.2% and over the past year the language's popularity has slowly declined[13]. Despite this, the language is still one of the most commonly used programming languages in the world.

Rank	Language	Share	1-year Trend
1	Python	28.05 %	+ 0.1%
2	Java	15.88 %	- 1.0 %
3	Javascript	9.27 %	- 0.3%

Table 1 - Most Popular Programming Languages (PYPL Index)

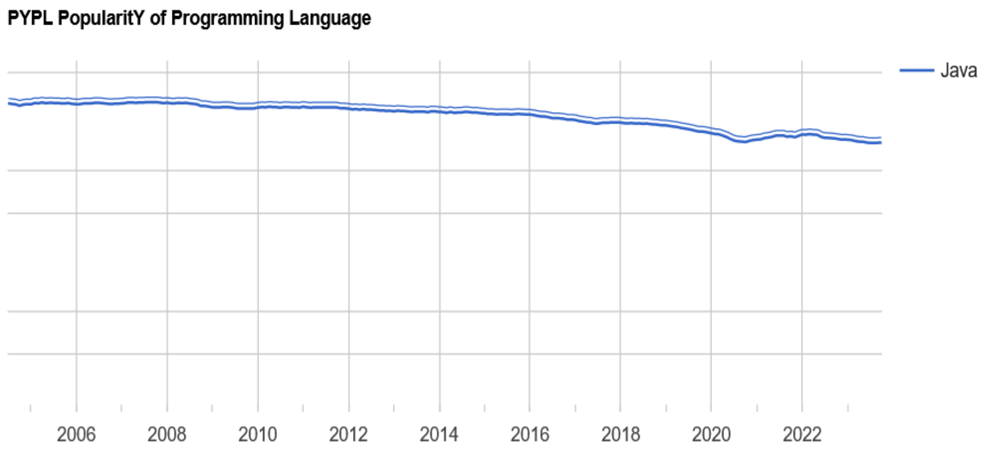


Figure 3 - Java's popularity throughout the years (PYPL Index)

The "*TIOBE Index"*[14], another ranking system that measures the popularity of programming languages based on search results, ranks Java, C/C++ and Python as the most popular programming languages in the world by October 2023. The index also shows that Java's use has declined by 3.92% compared to October 2022[14]. However, the language still remains one of the most widely used programming languages in the world.

Rank	Language	Ratings	Change
1	Python	14.82 %	- 2.25%
2	C	12.08 %	- 3.13 %

3	C++	10.67 %	+ 0.74 %
4	Java	8.92 %	- 3.29 %

Table 2 - Most Popular Programming Languages (TIOBE Index)

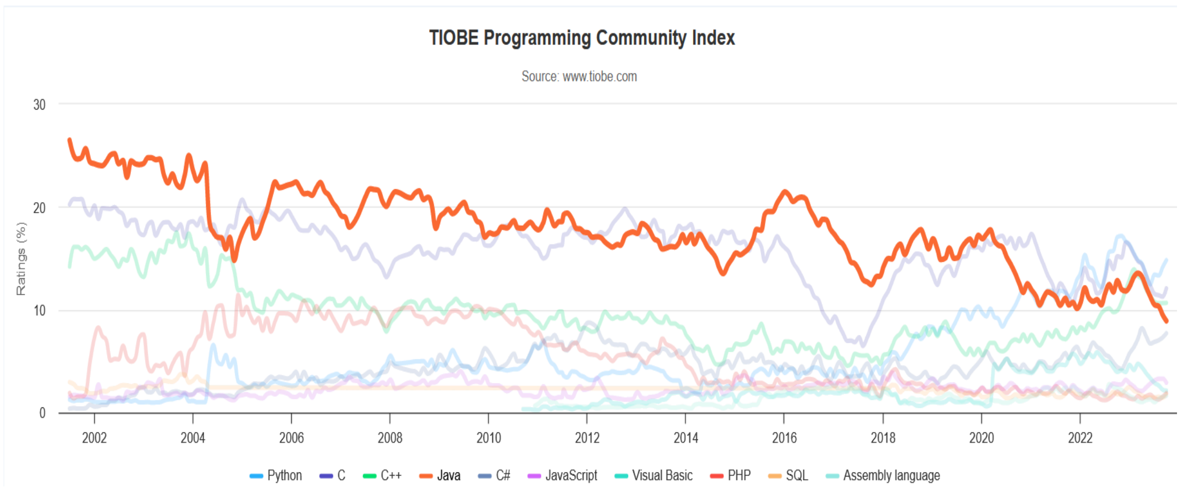


Figure 4 - Languages popularity throughout the years (TIOBE Index)

2.2 Software quality

Software quality is an important aspect of software engineering since it influences the overall success and reliability of software products. It includes a variety of factors that have a direct impact on the development processes as well as the end product's performance, maintainability, and user satisfaction. Understanding all aspects of software quality is necessary for building strong and maintainable software systems.

Software quality is a challenging concept to define precisely. As Kitchenham (1989) put it, quality is "*hard to define, impossible to measure, easy to recognize*"[15]. Additionally, Gillies states that, "*Quality is generally transparent when present, but easily recognized in its absence*"[16]. When software quality is present, it is visible in the software's performance, and ability to meet the requirements. The absence of software quality, on the other hand, becomes obvious when the program fails to fulfill expectations, exhibits bugs, or becomes difficult to maintain.

One critical aspect of software quality is maintainability, which refers to *"the extent to which software is capable of being changed after deployment"*[17], to fix errors that were not detected during the testing phase, improve performance, or adapt to changes in the requirements. Clean Code, a concept proposed by software engineer Robert C. Martin, encompasses a set of rules and principles aimed at producing understandable, maintainable, and efficient code. Finally, Technical Debt grows as software systems evolve over time and writing "clean" new code can be an efficient strategy for reducing TD, and thus preventing software decay over time[18].

2.3 Product Quality Model (ISO/IEC 25010)

The ISO/IEC 25010 standard, also known as the Software Quality Model[19], provides a framework for evaluating the characteristics of software products. It serves as an important tool for understanding and enhancing software quality. The model identifies the key quality attributes that should be considered during the evaluation process, thereby helping developers and stakeholders in making decisions about software development and deployment. Software quality attributes are non-functional requirements that can have an effect on the overall quality of a software product[20]. Many of these attributes can be addressed and evaluated during the time the architecture is developed. Software quality attributes include compatibility, usability, reliability and more[19]. The primary goal of the ISO/IEC 25010 quality model is to measure *"the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value"*[19].

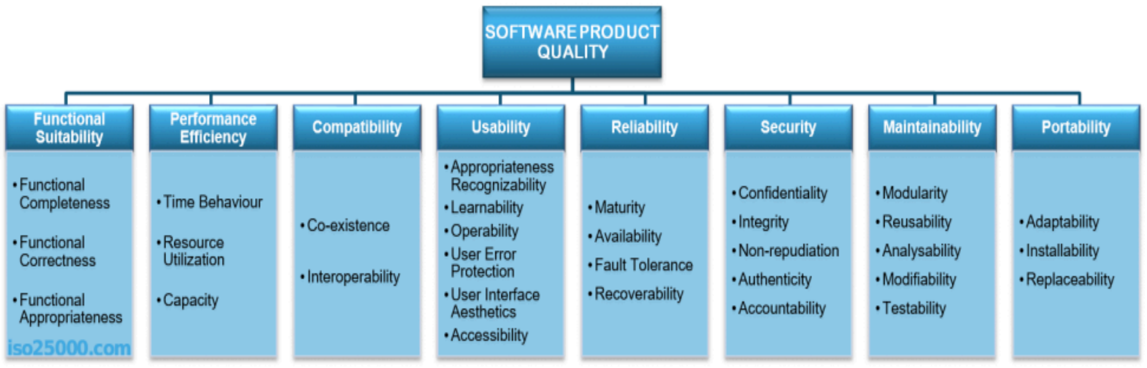


Figure 3 - Product Quality Model (ISO/IEC 25010)[1]

The product quality model is composed by eight quality attributes:

- **Functional Suitability**

This attribute represents the degree to which a software product provides functions that meet stated and implied needs when used under specified conditions. The attribute is composed of three attributes: Functional completeness, Functional correctness and Functional appropriateness[19].

- **Performance Efficiency**

This attribute represents the performance of the system, relative to the amount of resources used. Performance Efficiency is divided into 3 attributes: Time behavior, Resource utilization and Capacity[19].

- **Compatibility**

The degree to which a component can exchange information with other components, and perform its required functions while sharing the same hardware or software environment. This category is divided into 2 attributes: Co-existence and Interoperability[19].

- **Usability**

The degree to which a product or system can be used by specific users to achieve specific goals with effectiveness, efficiency and satisfaction in a specific context. The attribute is composed of the following sub-attributes: Appropriate recognizability, Learnability, Operability, User error protection, User interface aesthetics and Accessibility[19].

- **Reliability**

The degree to which a product performs specific functions under specified conditions for a specified period of time. Reliability is broken down into four categories: Recoverability, Fault tolerance, Availability and Maturity[19].

[1] Source: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

- **Security**

Security refers to the degree to which a product or system protects information and data so that persons or other products or systems have the appropriate degree of data access based on their types and levels of authorization. The attribute can be divided into the following categories: Authenticity, Accountability, Non-repudiation, Confidentiality and Integrity[19].

- **Maintainability**

This characteristic represents the degree of effectiveness and efficiency in which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements. Maintainability is composed of the following attributes: Testability, Modifiability, Analysability, Reusability and Modularity[19].

- **Portability**

Portability is the degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational environment to another. The attribute can be broken down into 3 sub-attributes: Replaceability, Installability and Adaptability[19].

2.4 Software Quality Tools

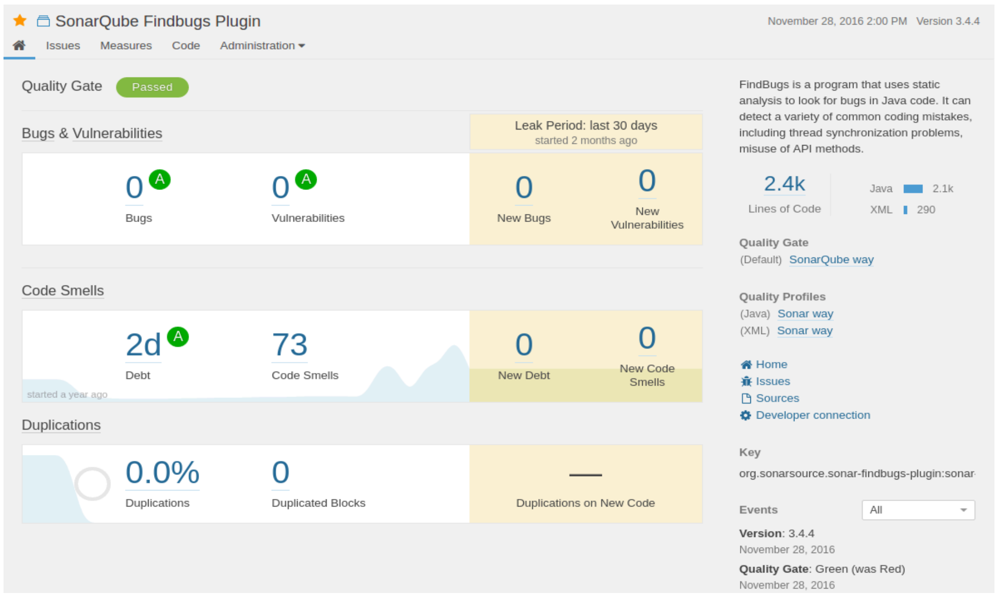
Software quality tools are critical components of the software development process because they provide automated tests and insights into code quality, maintainability, security, and coding standards conformance. These tools give developers crucial insights into their code, casting light on areas where changes and optimizations can be performed, thereby boosting the overall quality of the codebase.

The integration of software quality tools into the development process, particularly in the arena of Continuous Integration (CI) and Quality Gates, is one of the most important elements of these tools. Automated Analysis guarantees that quality is consistent and present throughout a project's lifecycle. This allows development teams to not only discover but also fix potential issues, long before they have the opportunity to become critical concerns that could jeopardize the system.

In essence, software quality tools are more than “just tools”, they are like partners in the development process. They advise developers in making appropriate decisions, and protect the codebase's integrity and security. Furthermore, they enable development teams to use automation to their advantage, reinforcing the development process and guaranteeing that the quality of their product is maintained from inception to deployment.

2.4.1 SonarQube

SonarQube[21] is an open-source, automatic code review tool that assists developers in producing clean and code smell free code. It provides a comprehensive set of quality guidelines for a variety of programming languages, including Java, Python, JavaScript, and others. SonarQube compares code to these guidelines and flags potential problems.


SonarQube Findbugs Plugin

November 28, 2016 2:00 PM Version 3.4.4

Issues Measures Code Administration

Quality Gate Passed

Bugs & Vulnerabilities

Leak Period: last 30 days
started 2 months ago

0 A Bugs
0 A Vulnerabilities
0 New Bugs
0 New Vulnerabilities

Code Smells

2d A Debt
73 Code Smells
0 New Debt
0 New Code Smells

started a year ago

Duplications

0.0% Duplications
0 Duplicated Blocks
— Duplications on New Code

FindBugs is a program that uses static analysis to look for bugs in Java code. It can detect a variety of common coding mistakes, including thread synchronization problems, misuse of API methods.

2.4k Lines of Code
Java 2.1k
XML 290

Quality Gate
(Default) SonarQube way

Quality Profiles
(Java) Sonar way
(XML) Sonar way

Home
Issues
Sources
Developer connection

Key
org.sonarsource.sonar-findbugs-plugin:sonar

Events All

Version: 3.4.4
November 28, 2016

Quality Gate: Green (was Red)
November 28, 2016

Figure 4 - A SonarQube project homepage[2]

One of the key features of SonarQube is its ability to detect code smells or design flaws that can lead to maintainability problems, while also identifying security vulnerabilities to fortify software against potential threats. Additionally, SonarQube highlights code duplications, enabling developers to eliminate redundant code. The platform can also integrate with Continuous Integration (CI) and Continuous Deployment (CD) pipelines to ensure code quality is maintained throughout the whole development

[2] Source: https://en.wikipedia.org/wiki/SonarQube

process. Developers can discover bugs early and prevent them from spreading to the final product by automating code analysis within the CI/CD workflow.

2.4.2 UoM Dashboard

UoM Quality Dashboard[3] is a project developed by the University of Macedonia. implemented by two University teams: OpenSource UoM[23] and Software Engineering Lab UoM[24]. The primary purpose of the project is to create an all-encompassing dashboard that efficiently presents data related to numerous projects managed by the University's Applied Informatics Department. In essence, this dashboard is a sophisticated visualization tool that has been built to assist organizations in monitoring and examining the quality of their software activities.

UoM
DASHBOARD
A project by OpenSource UoM, SE UoM and University of Macedonia

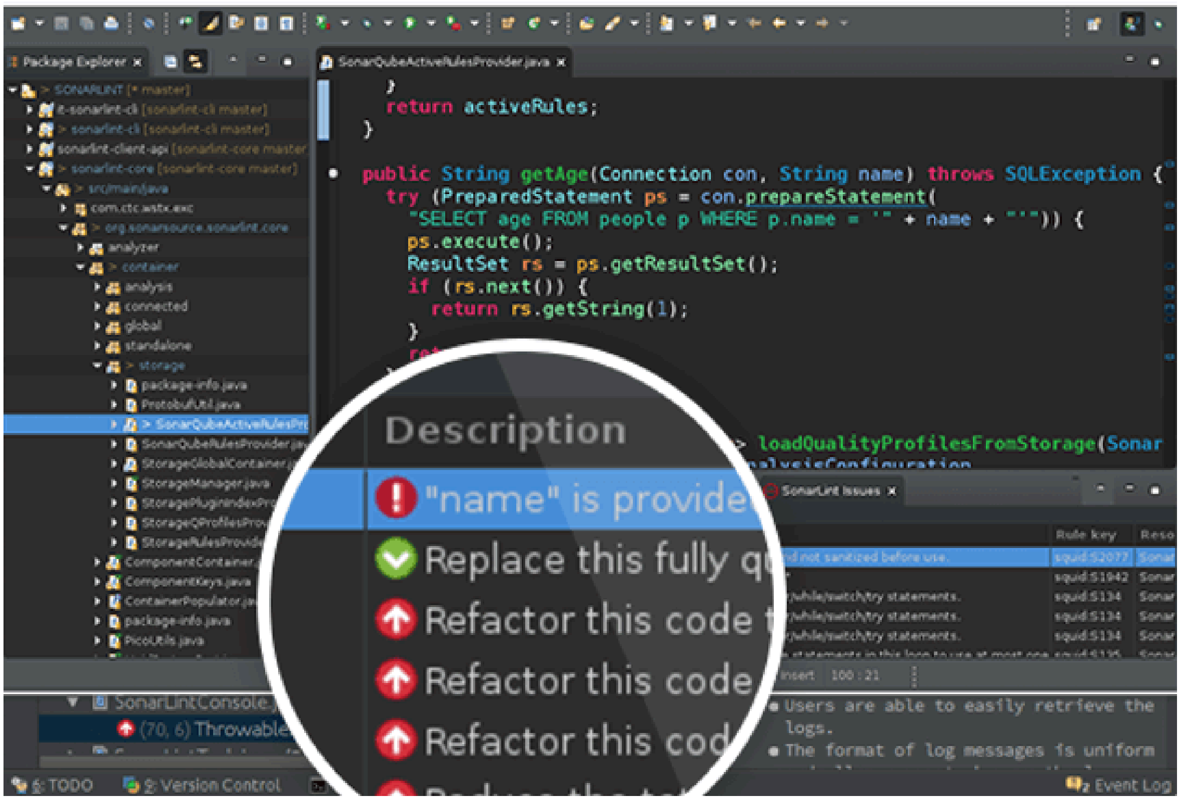
The dashboard acts as a powerful analytical tool while promoting a culture of healthy competition inside organizations. It encourages a dynamic and motivating environment where teams can strive for excellence and continuously improve their software projects by providing means to evaluate and compare the performance of developers based on their activity and overall code quality. This feature provides value to the project by encouraging a spirit of creativity and quality in software development. The tool ingeniously combines software quality data and repository statistics in its analyses, providing useful insights into the development process. The dashboard delivers a holistic and uniform analysis of all the repositories examined by effectively integrating data analysis from multiple third-party services such as SonarQube[21], PyAssess[25], and

[3] https://github.com/SE-UoM/quality-dashboard

CodeInspector (which is the proposed tool developed in this thesis). The tool is conveniently divided into numerous displays, each dedicated to a different data category, providing a wide range of information, from general data such as total projects and files and insights into the most popular programming languages to various aspects of project quality, such as code smells and technical debt. While the project is still in its early stages, it is always evolving to deliver more profound and informative information.

2.4.3 SonarLint

SonarLint[26] stands as a cutting-edge, state-of-the-art code analysis tool, engineered to seamlessly integrate within a wide variety of Integrated Development Environments (IDEs) It provides real-time code analysis while developers create code, delivering immediate feedback on potential flaws and making suggestions for changes. SonarLint works with a variety of IDEs, including Visual Studio Code, IntelliJ IDEA, Eclipse, and others, making it accessible to developers working in a variety of environments.



```
    }
    return activeRules;
  }

  public String getAge(Connection con, String name) throws SQLException {
    try (PreparedStatement ps = con.prepareStatement(
      "SELECT age FROM people p WHERE p.name = '" + name + "'")) {
      ps.execute();
      ResultSet rs = ps.getResultSet();
      if (rs.next()) {
        return rs.getString(1);
      }
```

Description
"name" is provide
Replace this fully q
Refactor this code
Refactor this code
Refactor this cod

Figure 4 - SonarLint[4]

The tool employs the same set of static code analysis rules as SonarQube, ensuring quality assessment consistency throughout the whole development team. SonarLint allows developers to fix code smells, bugs, and security vulnerabilities as they write code, minimizing the time and effort necessary for subsequent code reviews. The tool helps developers adopt better coding habits and maintain a high level of code quality from the start by integrating smoothly within the development environment. This extraordinary

[4] Source: https://www.tatvasoft.com/blog/introduction-to-sonarqube-sonarlint/

plugin is more than just an addition to the development workflow and is a true defender of code quality and integrity. As developers write their code, sonarlint is always by their side, performing real-time analysis, not just as a passive observer but as an active collaborator, ready to provide quick feedback on problems or ideas for changes.

2.4.4 CodeClimate

CodeClimate[27] is a cloud-based platform that provides automated code review and analytics for software projects. It supports multiple programming languages, including Ruby, Python, and more.

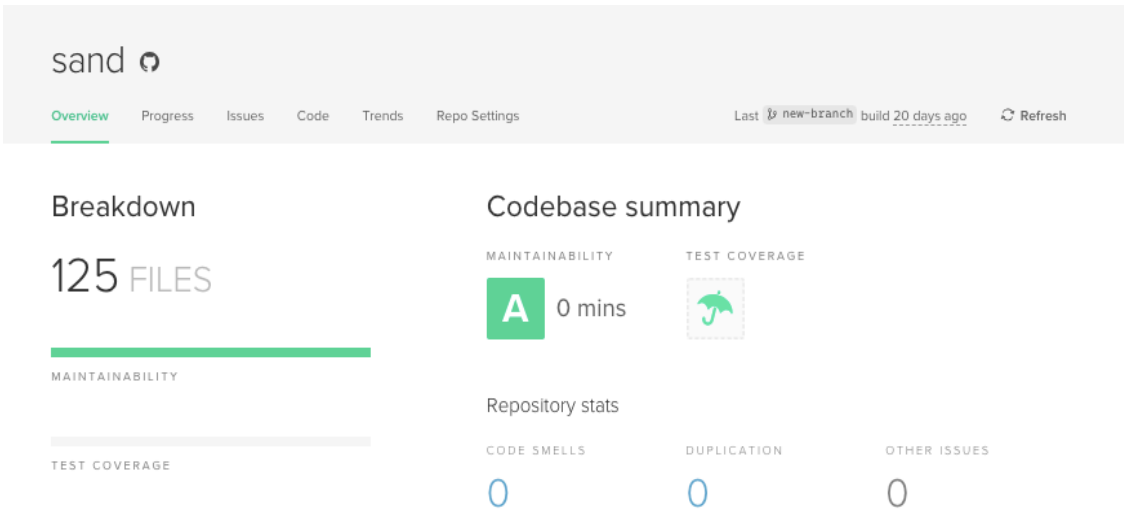
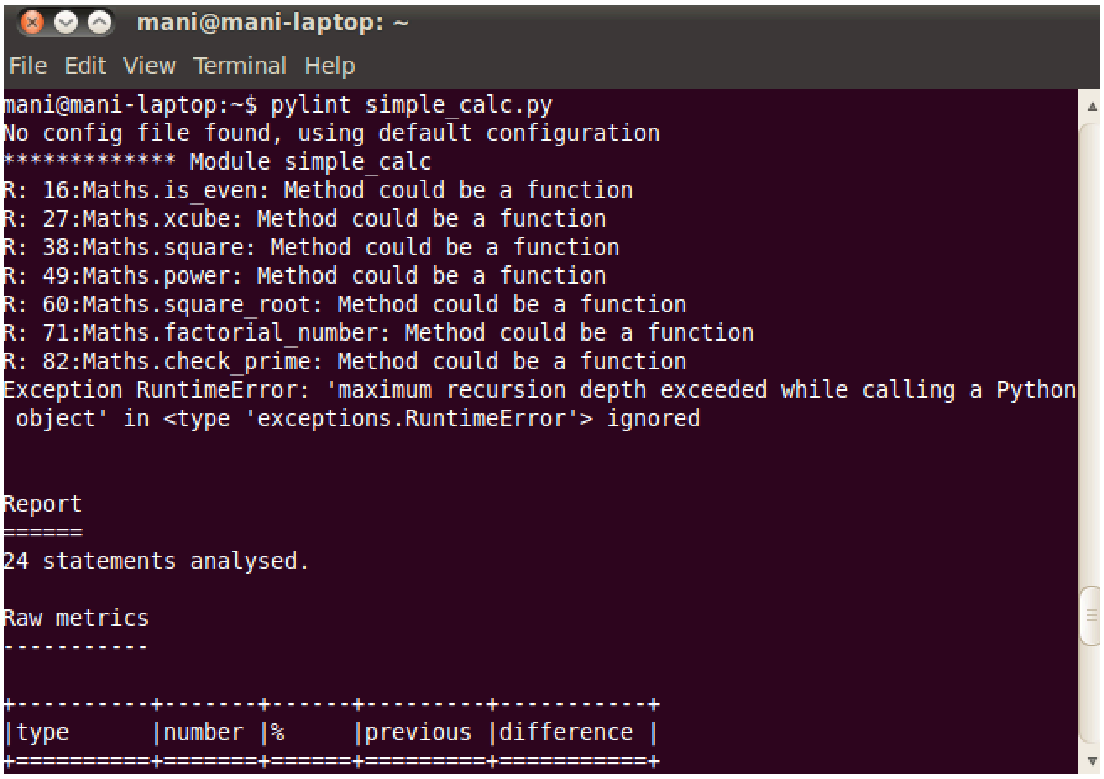


Figure 5 - Code Climate Home Page

The tool analyzes code repositories and determines maintainability scores, assisting programmers and quality assurance analysts in identifying areas for improvement. It detects code smells, complexity concerns, and duplication, allowing teams to keep their codebases clean and maintainable. The platform also evaluates test coverage, revealing information about the effectiveness of test suites. Integrating Code Climate with CI/CD pipelines enables teams to continuously check code quality and guarantee that each code change adheres to quality requirements. Teams may maximize code quality, improve code maintainability, and streamline the development process using Code Climate's insights.

2.4.5 PyLint

Pylint[X] is a tool for analyzing Python codebases. It includes a plethora of tools to assist developers in maintaining clean and efficient Python code.



```
mani@mani-laptop:~$ pylint simple_calc.py
No config file found, using default configuration
************* Module simple_calc
R: 16:Maths.is_even: Method could be a function
R: 27:Maths.xcube: Method could be a function
R: 38:Maths.square: Method could be a function
R: 49:Maths.power: Method could be a function
R: 60:Maths.square_root: Method could be a function
R: 71:Maths.factorial_number: Method could be a function
R: 82:Maths.check_prime: Method could be a function
Exception RuntimeError: 'maximum recursion depth exceeded while calling a Python
 object' in <type 'exceptions.RuntimeError'> ignored


Report
======
24 statements analysed.

Raw metrics
-----------

+----------+-------+------+---------+----------+
|type      |number |%     |previous |difference |
+==========+=======+======+=========+===========+
```

Figure 6 - PyLint Example

PyLint performs static code analysis to detect code smells, security vulnerabilities, and potential errors. Using Pylint in the development processes allows developers to obtain fast feedback on the quality of their code and make changes early in the development cycle. Pylint's insights and recommendations promote continuous improvement by encouraging the team to write cleaner, more maintainable code. Moreover, Pylint can also integrate with CI/CD pipelines, enabling automated code analysis throughout the development lifecycle. This integration ensures that code quality is continuously monitored, reducing the risk of introducing new issues into the codebase.

2.4.6 Automating Quality Analysis

Automating Quality Analysis has become a vital part of modern software development practices, influencing quality assurance and the adherence to consistent standards, both of which are essential for a software project's success. Automation speeds up the development process, increases team productivity, and reduces the possibility of human error. In this context, Continuous Integration (CI) and Quality Gates stand out as two valuable parts of this approach that work together to simplify code analysis and enable early fault discovery. All things considered, the combination of Quality Gates, Continuous Integration (CI), and automation not only speeds up the development process but also establishes a culture of strict quality control from the very beginning of the project. Thus guaranteeing that the final product is software that is reliable, safe, and compliant with the highest quality standards.

2.4.6.1 Continuous Integration (CI)

Continuous Integration is a software development practice where members of a team integrate their work frequently, usually at least daily, resulting in numerous integrations per day[29]. The primary objective of CI is to enable developers to detect and address integration issues as early as possible. Every integration initiates a set of automated builds tests, ensuring the codebase's continuous functionality and reliability. With automation being the driving force, this approach not only promotes cooperation but also creates a strict schedule for quality evaluation.

When a code change is submitted to the repository, automated build and testing procedures are triggered as part of the CI process. These automated methods build the code, test it, and analyze it to find any problems. CI ensures that the main codebase remains stable at all times by validating each code change in an isolated environment.

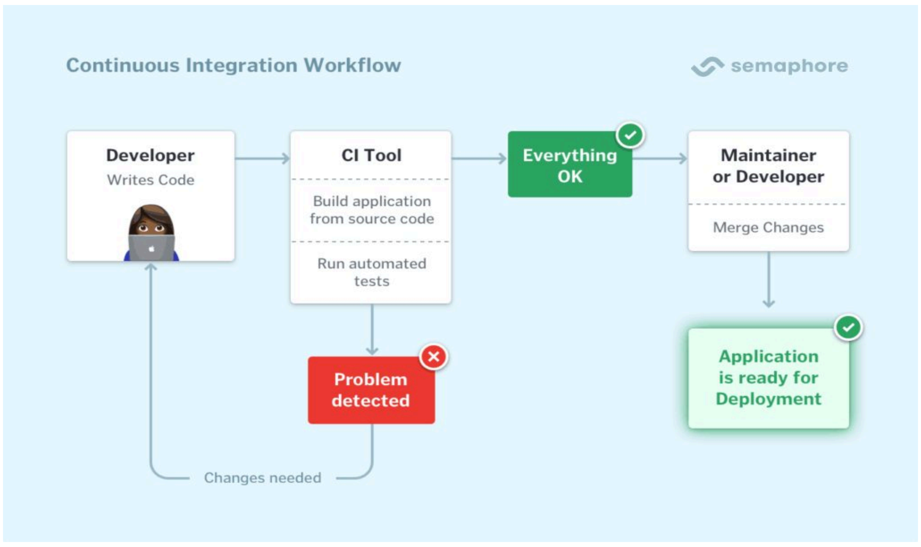


Figure 7 - Continuous Integration Workflow[5]

2.4.6.2 Quality Gates

Quality Gates[30] are a set of predefined criteria that must be met by code changes before they can be merged into the main codebase or sent to production. These criteria include a wide range of attributes in terms of the software quality of the product, including readability, maintainability, security, and performance.

An essential component in ensuring that these standards are met is the Continuous Integration (CI) system. When a developer submits a code change or a pull request the CI system initiates a sequence of automated tests and evaluations of the code. These tests investigate many aspects of quality to make sure the code update satisfies the established standards. If the code change satisfies all of the Quality Gate requirements it is seamlessly integrated into the main codebase. However, if any of the Quality Gate criteria are not met by the change, the CI system rejects it and prevents it from being incorporated into the main codebase.

As an illustration of Quality Gates in action, let's look at SonarQube. SonarQube has the feature to establish and enforce Quality Gates and will stop the merge or deployment process if a code change breaches any of the established quality criteria, thus preserving the integrity of the codebase. This contributes to the delivery of more

[5] Source: https://semaphoreci.com/continuous-integration

maintainable software by guaranteeing that only code that satisfies the predetermined quality requirements is permitted into the main codebase or production environment.

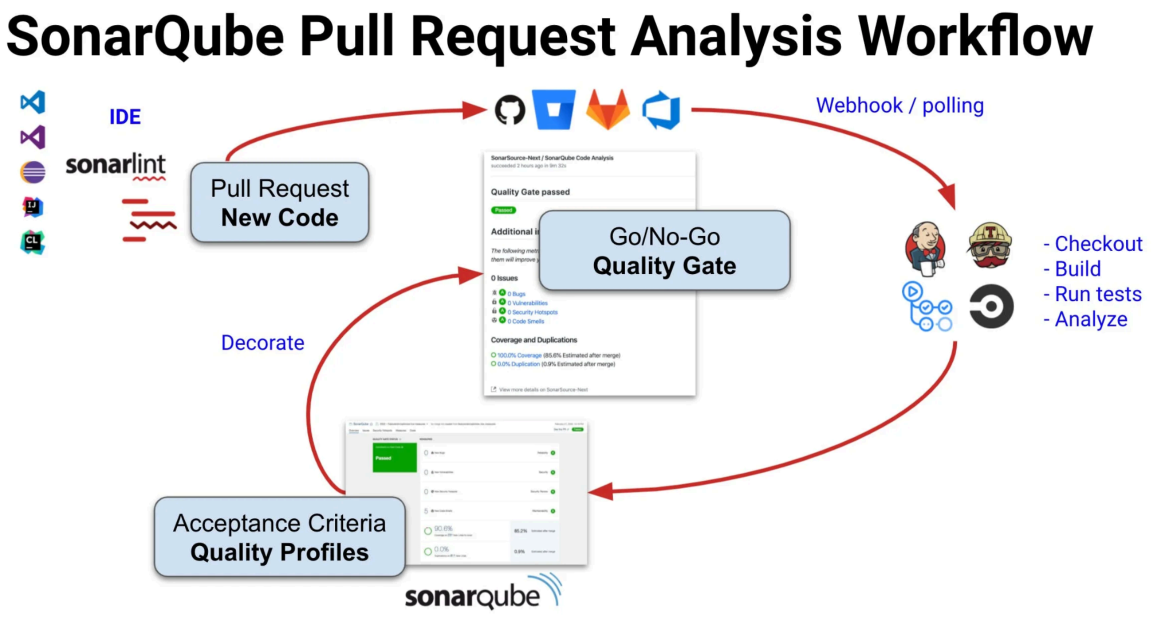


Figure 8 - SonarQube Quality Gates Workflow[6]

[6] Source: https://www.sonarsource.com/blog/clean_coding-quality_profile_quality_gate_guidance/

2.5 The Delta Maintainability Model (DMM)

The Delta Maintainability Model (DMM)[31] is a software metric that assesses the maintainability of code changes in a software system. The model was published by M. di Biase et al. in the 2019 IEEE/ACM International Conference on Technical Debt.

2.5.1 Conceptual Framework

The model is based on determining the impact of an individual code change on maintainability. It views changes as the addition or removal of lines of code (LOCs) to units and modules implicated in the change (similar to a diff). A high DMM score indicates a large number of complex alterations, which may have an negative effect on the system's overall maintainability[31]. The DMM utilizes the SIG-MM [32] system properties as its foundation. These properties encompass duplication, unit size, unit complexity, unit interfacing, module coupling, and file coupling[32] and each property is defined with a specific description and criteria for qualifying code as low risk.

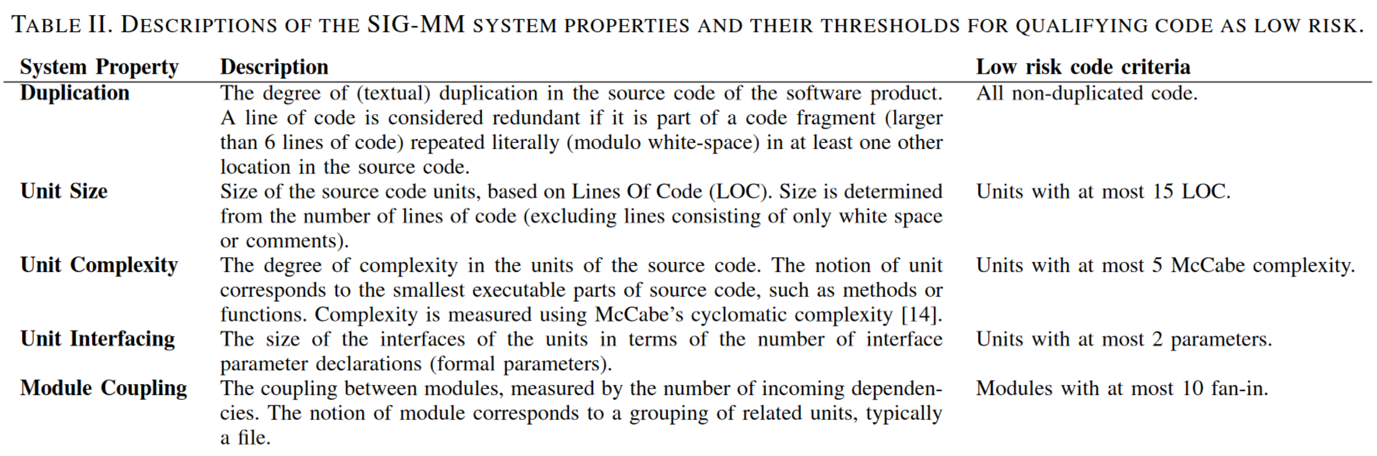


TABLE II. DESCRIPTIONS OF THE SIG-MM SYSTEM PROPERTIES AND THEIR THRESHOLDS FOR QUALIFYING CODE AS LOW RISK.

System Property	Description	Low risk code criteria
Duplication	The degree of (textual) duplication in the source code of the software product. A line of code is considered redundant if it is part of a code fragment (larger than 6 lines of code) repeated literally (modulo white-space) in at least one other location in the source code.	All non-duplicated code.
Unit Size	Size of the source code units, based on Lines Of Code (LOC). Size is determined from the number of lines of code (excluding lines consisting of only white space or comments).	Units with at most 15 LOC.
Unit Complexity	The degree of complexity in the units of the source code. The notion of unit corresponds to the smallest executable parts of source code, such as methods or functions. Complexity is measured using McCabe's cyclomatic complexity [14].	Units with at most 5 McCabe complexity.
Unit Interfacing	The size of the interfaces of the units in terms of the number of interface parameter declarations (formal parameters).	Units with at most 2 parameters.
Module Coupling	The coupling between modules, measured by the number of incoming dependencies. The notion of module corresponds to a grouping of related units, typically a file.	Modules with at most 10 fan-in.

Figure 9 - SIG-MM Properties and Thresholds for qualifying code as Low Risk[7]

[7] Source: https://ieeexplore.ieee.org/document/4335232

2.5.2 Model Calculation

The calculation of the DMM consists of two levels:

1. **Risk Profile Mapping:** This level describes how a code change translates into a Risk Profile - a concept originally derived from SIG-MM but adapted for DMM [31].
2. **DMM Score Generation:** This level combines all Risk Profiles for a code change to generate a DMM score[31].

The model assesses and categorizes the five code properties (Figure 2) into a *Risk Profile* (low, medium, high, very-high) for each file changed within a commit and the original files before the modification. Following that, the model computes various *Deltas*, which are the differences in lines of code measured before and after a file update.

To aggregate Deltas at commit level:

- The model sums all the Delta values for each Code Property, resulting in every commit having two values per Risk Profile (one for Increases, one for Decreases).
- Increases in LOC in low risk profiles are considered good and highly-maintainable changes, along with decreases in LOC in medium, high and very-high categories.
- On the other hand, decreases in LOC in the low risk category as well as increases in LOC in medium, high and very-high categories are considered harmful to maintainability.

Finally, the *Delta Score* for each Code Property is determined as a fraction of the total highly maintainable Low Risk Profile Delta. When the five Code Properties are added together, the result is known as the *Delta Maintainability Score* (DMM Score). Both *Delta Score* and *DMM Score* have a value between 0.0 and 1.0, with 0.0 representing the lowest maintainability and 1.0 being the highest[31].

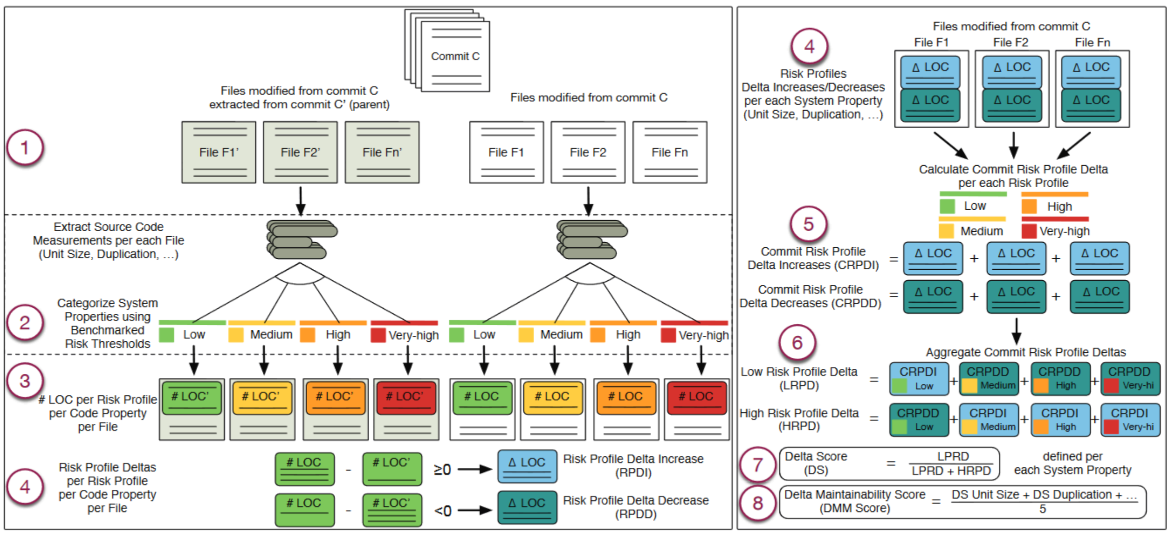


Figure 10 - Overview of the DMM[8]

2.5.3 DMM in PyDriller

In the proposed tool, PyDriller is a key component to extract information from GitHub repositories, such as complexity and churn and DMM metrics. PyDriller provides an implementation of the Open Source Delta Maintainability Model (OS-DMM) to assess the maintainability implications of commits. This implementation provides a solution suited for research experiments and measurements for systems developed in common programming languages. While PyDriller's git functionality is language-independent, the metrics it generates (such as method size and cyclomatic complexity) require language-specific implementations, on which PyDriller relies on Lizard.[33]

The OS-DMM implementation of PyDriller supports three commit-level metrics related to risk in size, complexity, and interfacing[33]. In essence, "*the delta-maintainability metric is the proportion of low-risk change in a commit. The resulting value ranges from 0.0 (all changes are risky) to 1.0 (all changes are low risk). It rewards making methods better, and penalizes making things worse*"[33]. As mentioned in the previous section the starting point for the DMM is a risk profile. Usually, risk profiles divide units into four categories: **low**, **medium**, **high**, and **very high** risk methods. A class's risk profile is then a quadruple representing the amount of code in each of these four areas.

[8] Source: https://ieeexplore.ieee.org/document/8785997

To make things simpler, PyDriller uses only two categories: low risk, and non-low risk code changes (medium, high, or very high changes). The library uses Delta risk Profiles to shift risk profiles from the system level to the commit level. This is a pair (dl, dh), with **dl** representing an increase/decrease in low risk code and **dh** representing an increase/decrease in high risk code[33].

This Delta Risk Profile can be used to identify positive and negative change:

- Increases in low risk code are considered beneficial, whereas increases in high risk code are considered harmful[33].
- Decreases in high risk code are considered positive, but decreases in low risk code are negative unless no high risk code is introduced to the codebase[33].

The final DMM values are then calculated using the following formula[33]:

$$\frac{good\ change}{good\ change + bad\ change}$$

Figure 11 - PyDriller DMM value calculation

2.6 Prioritization Techniques in Code Quality Assessment

In software quality assurance, the need for efficient prioritization of tasks and resources is vital. Various methodologies and tools have been used to streamline the process of prioritizing tasks, one of the most popular being “The Eisenhower Matrix”[34]. The Eisenhower Matrix, also known as the “Urgent-Important Matrix”, is a time management model that categorizes tasks into four quadrants based on how urgent and important they are. This matrix serves as a practical tool for individuals to prioritize their tasks and make more informed decisions about where to allocate their time and effort. The essence of the Eisenhower Matrix is to classify tasks in four quadrants:

- **Urgent and Important (Quadrant I - DO)**
  Tasks that demand immediate attention due to their high urgency and importance.

- **Important but Not Urgent (Quadrant II - DECIDE)**
  Tasks that are important for long-term goals and success but may not require immediate action.
- **Urgent but Not Important (Quadrant III - DELEGATE)**
  Tasks that are pressing but may not contribute significantly to one's overall goals so they can be delgated.
- **Not Urgent and Not Important (Quadrant IV - DELETE)**
  Tasks that are neither urgent nor important and can often be eliminated.

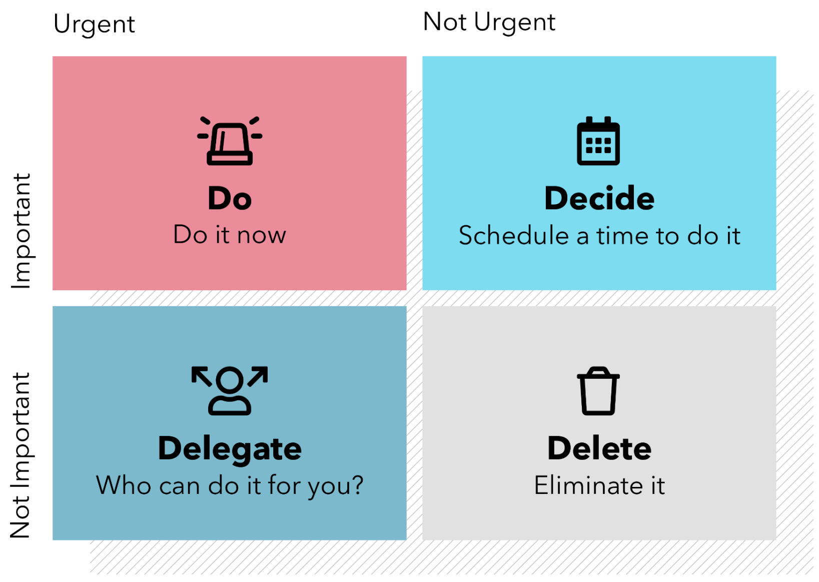


Figure 12 - The Eisenhower Matrix[9]

By applying a similar approach for our Hotspot Analysis, we can easily prioritize "hotspot" files within a codebase using code complexity (cc) and churn (number of changes) as the dimensions for categorizing the files in a repository. Our proposed code quality matrix operates as follows:

- **High Cyclomatic Complexity and High Churn (High Priority)**
  Files with both high code complexity and high number of changes are considered “High Priority” due to their complexity and frequent modifications.

[9] Source: https://www.spica.com/blog/the-eisenhower-matrix

- **Low Cyclomatic Complexity and Low Churn (Low Priority)**
  Files with low code complexity and few changes are categorized as "Low Priority". These files are not pressing issues and probably will not significantly impact the codebase's quality.
- **Low Cyclomatic Complexity and High Churn (Normal Priority)**
  Files with low code complexity but a high number of changes fall into the "Normal Priority" category. These files are not inherently complex, but their frequent modifications suggest that they need regular attention.
- **High Cyclomatic Complexity and Low Churn (Medium Priority)**
  Files with high code complexity but low churn are deemed as medium priority since, although these files are complex, they are not modified that often.

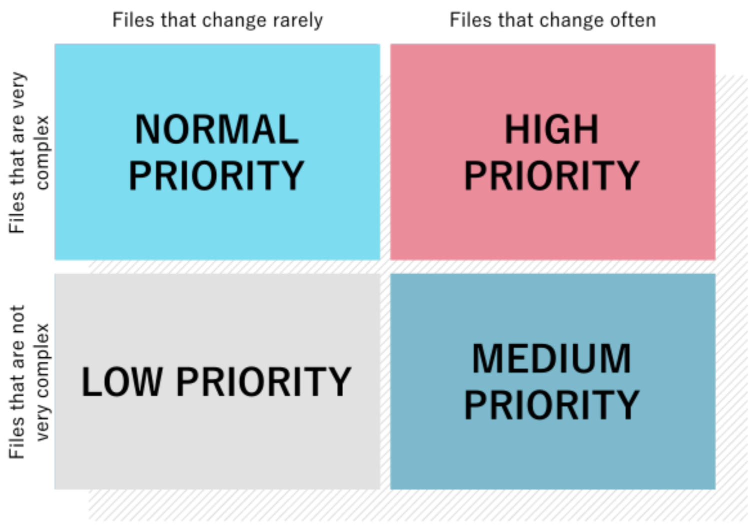


Figure 13 - Our Quality Matrix

3 Evaluation Tool - “CodeInspector”

The proposed tool, named ***CodeInspector*** is a full-stack web application that aims to analyze code written in multiple programming languages. It extracts information from GitHub repositories, performs hotspot identification based on complexity and churn metrics, and uses the Delta Maintainability Model (DMM) to assess the maintainability impact of individual code changes (commits).

3.1 Tech Stack

3.1.1 Python (Programming Language)

Figure 14 - Python Logo

Python, a high-level, versatile programming language, forms the core of CodeInspector's backend infrastructure. The language was developed by Guido van Rossum and first released in 1991[35]. It prioritizes code readability through the use of indentation[36] and the syntax is quite similar to the English language, thus making it easy to learn and offering clean, human-readable code. Python's popularity extends beyond its use as a computer language. According to the 2023 Stack Overflow Developer Survey[9], Python ranks as the third most popular programming language across all surveyed developers. Its applications include data analytics, DevOps, web crawling, web server development, and other areas. Python's ease of use and simple syntax also make it especially popular among beginning programmers.

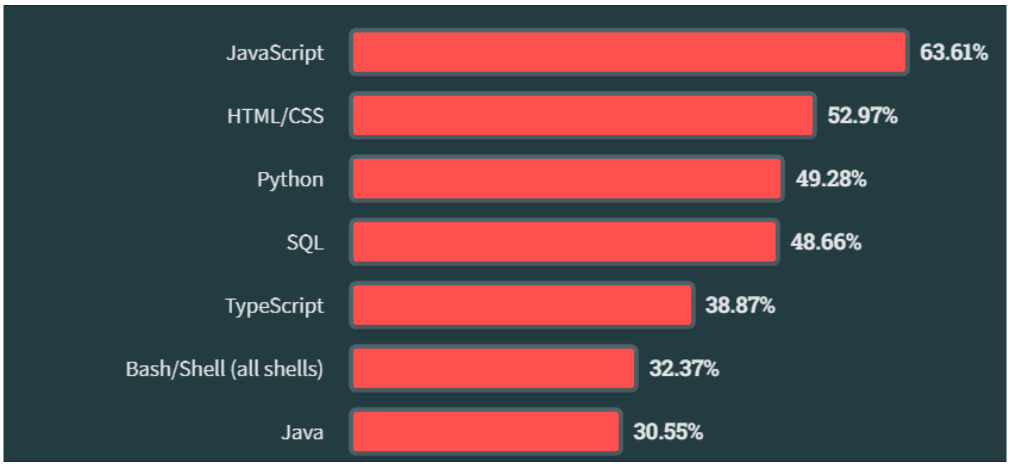


Figure 15 - Stack Overflow Developer Survey 2023

Python's active community provides a steady supply of updates and improvements. Python 3.12[37] is the most recent version as of October 2023. The language is a fantastic choice for a wide range of disciplines, notably data science and machine learning, thanks to its lively development environment and active community and its vast ecosystem significantly improves our evaluation tool by providing a diverse set of libraries and tools for data manipulation and statistical analysis. NumPy[38], VisPy[39] and Pandas[40] are some examples of popular data analysis libraries. These libraries include comprehensive data structures and embedded mathematical operations, allowing for fast computations while accommodating multidimensional data and large matrices.

In conclusion, Python's simplicity, versatility and active community make it a perfect choice not only for CodeInspector but also for a wide range of applications in data science and beyond.

3.1.2 FastAPI

FastAPI

Figure 16 - FastAPI Logo

FastAPI is a *"modern, fast (high-performance), web framework for building APIs with Python 3.8+ based on standard Python type hints"*[41]. The framework brings a plethora of capabilities to the table such as automatic data validation, serialization, and the development of API documentation. This combination ensures that incoming requests are not only appropriately formed but also contain all of the necessary data, considerably lowering the possibility of data-related errors and increasing the overall durability of our tool. The asynchronous capabilities of FastAPI make it a good choice for handling concurrent API requests, which is important when dealing with large data volumes, such as analyzing GitHub repositories. This capability optimizes response times and guarantees the tool's responsiveness even when dealing with resource-intensive tasks while the simultaneous management of multiple requests makes sure that the tool remains efficient, even under heavy workloads.

Another benefit of FastAPI is its interactive API documentation[42], which is enabled by Swagger UI[43]. This user-friendly interface allows developers to easily explore the numerous API endpoints and comprehend their functions. Documentation for APIs is a useful resource for understanding the system's capabilities and efficiently interacting with it, hence speeding the development process. In summary, FastAPI provides us with the tools and features we need to build a strong, high-performance REST API with asynchronous capabilities and user-friendly documentation, making it an excellent solution for our project's requirements.

3.1.3 React JS

Figure 17 - React.js Logo

React JS[44], a popular JavaScript library, is responsible for shaping the front-end user interface of the tool. Through its component-based architecture, the library enables developers to create a variety of reusable and modular UI components, providing a streamlined and scalable development process that not only speeds up the creation of aesthetically pleasing interfaces but also simplifies maintenance and updates.

The library’s versatility extends to its ability to present analytical results for users with efficiency. React JS ensures that data is delivered to consumers with speed and fluidity thanks to its powerful rendering engine and state management. Furthermore, its ability to fluidly manage user interactions raises user experience, making it intuitive and interactive. React JS enables users to interact with the evaluation tool easily by offering real-time feedback and dynamic updates, resulting in a more engaging and fulfilling experience. The tool’s interface is powerful and user-friendly, simplifying complex processes and promoting rapid decision-making.

3.1.4 Docker

Figure 18 - Docker Logo

Docker[45] plays a big role in the evaluation tool's deployment and portability. It enables the encapsulation of the application and its dependencies into lightweight, isolated containers. This approach ensures consistent environments across various systems, streamlining the deployment process and minimizing compatibility issues. The platform provides consistency across a wide range of systems. This consistency simplifies and speeds up the deployment process by ensuring that the application runs consistently across platforms. Docker eliminates the need to deal with compatibility issues by encouraging a fluid and harmonious experience across multiple environments. Moreover, it allows us to package our evaluation tool as a standalone container, immune to the complexities of the host system. When dockerized, our tool becomes a dependable and predictable entity, ready to perform consistently regardless of the surrounding technical details.

3.1.5 MySQL

Figure 19 - MySQL Logo

To efficiently preserve and manage data within our tool, we have incorporated a MySQL database, which provides a solid and structured foundation for both the storage and retrieval of crucial data. MySQL[46] is a popular open-source relational database management system renowned for its speed, and scalability. Its incorporation into our system improves data management and maintains data consistency. MySQL allows the enforcement of restrictions such as primary and foreign keys to preserve data integrity. With MySQL we can ensure the precision and coherence of stored data while lowering the chance of mistakes and data corruption.

As our tool grows and handles larger datasets, the scalability characteristics of MySQL enable us to easily change the database to meet new demands, ensuring that our tool remains efficient as requirements advance. MySQL is built for speed and efficiency, using techniques like indexing and caching to deliver fast data retrieval and processing, resulting in faster response times and better user experiences. It also has strong security measures, such as user authentication, access control, and encryption choices, to protect sensitive data and prevent unauthorized access and data breaches.

Furthermore, MySQL is well-known for its fault tolerance and dependability, including mechanisms for data backup and replication, that ensures data availability even in the face of hardware failures or other challenges. MySQL supports a variety of data formats, making it interoperable with a large range of applications and tools, allowing for simple data exchange and interaction with other systems. On top of that, MySQL has a big and active user community, and provides access to a variety of documentation, tutorials, and third-party tools, as well as professional support services for enterprise-level applications. Using MySQL within our evaluation tool not only improves data management and storage capabilities, but it also establishes a solid foundation for scalability, performance, security, and data integrity, eventually contributing to a better user experience and a more secure environment.

3.1.6 PyDriller

PyDriller[47] is a Python framework that simplifies mining software repositories. It enables easy extraction of various code-related data from Git repositories, including commits, developers, modifications, diffs, and source codes. PyDriller's capabilities facilitate the gathering of essential metrics required for evaluating code quality, such as code churn, complexity, and maintainability. With PyDriller, the evaluation tool can access the entire history of a software project stored in a Git repository. This allows developers to analyze the evolution of code quality and hotspots over time, identifying patterns and trends that can inform decision-making. As of August 21, 2023, the latest version of PyDriller is 2.5.11, and it can be conveniently installed using the 'pip' package manager[47], ensuring that users have access to the most up-to-date features and enhancements offered by this framework. PyDriller's capabilities make it a valuable resource for those seeking to unlock insights hidden within software repositories and drive data-informed decision-making processes.

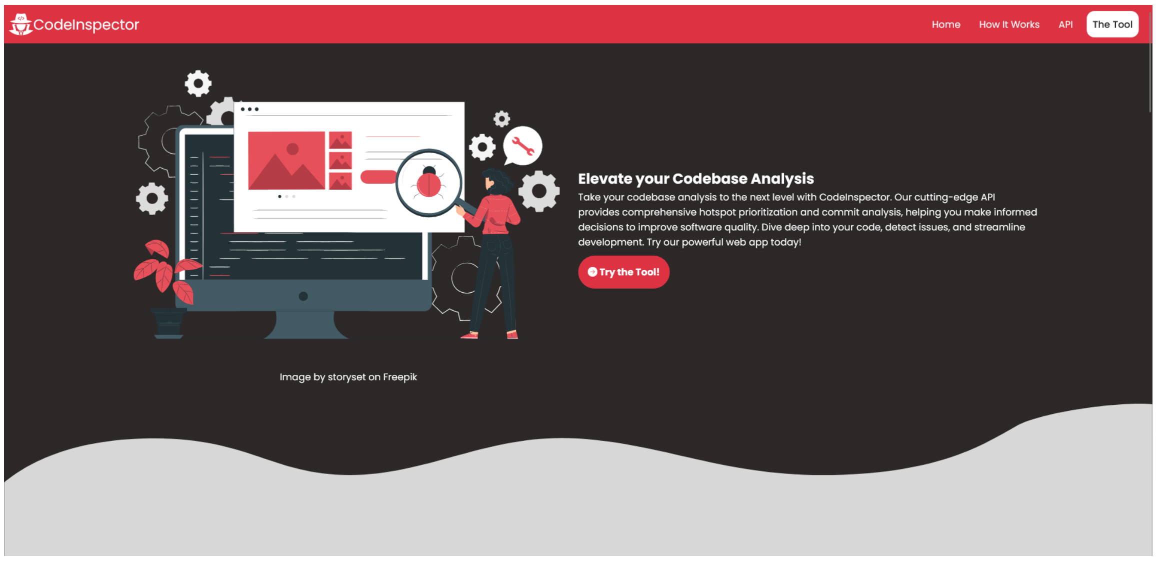
3.2 Tool Functionality

Before we dive into the technical implementation of the tool, it is crucial to understand its functionalities, which form the backbone of the user experience. The primary functionality of the tool centers around code analysis and visualization. Leveraging various metrics, the app examines the repository's codebase, identifying potential hotspots and assessing the impact of individual commits. In addition to its functionalities, "Code Inspector" is structured as a well-orchestrated system, divided into front-end and backend components. These components work together to ensure a smooth user experience. The frontend is responsible for presenting the user interface, allowing developers to interact with the tool easily while the backend handles the heavy lifting of data processing and code analysis.

As we get into the technical aspects of its implementation, the following chapters will shed light on the underlying architecture and techniques, all of which contribute to delivering a seamless and empowering user experience.

3.2.1 Home Page (Front End)

The home page serves as the gateway to the "CodeInspector" application. It presents users with a friendly interface and essential sections to introduce and navigate the tool. At the top of the Home Page (as with all other pages) there is a navigation bar that allows users to access different pages of the tool with ease. The user is welcomed with the Tool Presentation section, where the tool is introduced, showcasing its capabilities and benefits to users. An inviting "Try the Tool" button encourages users to access the Tool Page. Following the Tool Presentation section we have the Features Section highlighting the core functionalities that make the Code Inspector stand out.



Elevate your Codebase Analysis

Take your codebase analysis to the next level with CodeInspector. Our cutting-edge API provides comprehensive hotspot prioritization and commit analysis, helping you make informed decisions to improve software quality. Dive deep into your code, detect issues, and streamline development. Try our powerful web app today!

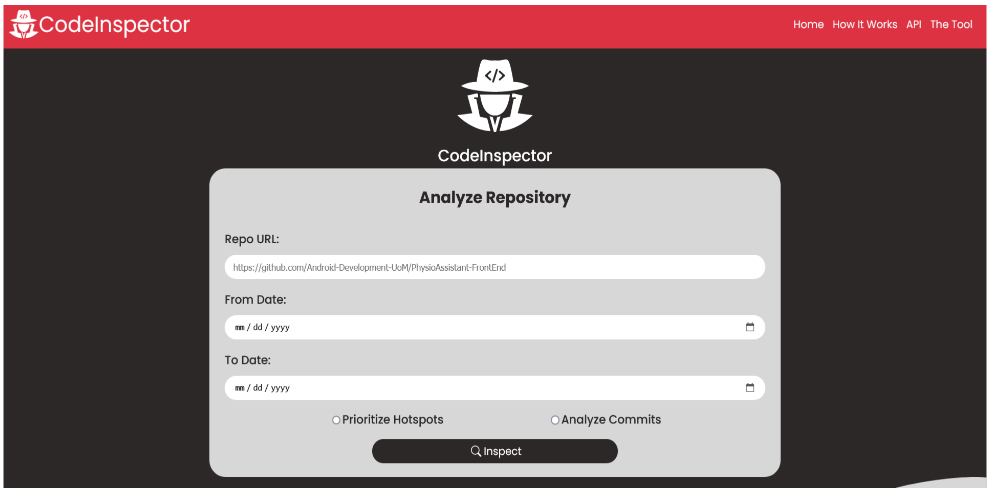
Figure 20 - CodeInspector Homepage (Tool Presentation)

3.2.2 Tool Page

The Tool Page serves as the heart of the Code Inspector, offering a user-friendly interface for developers to perform analyses. At the core of the page is the Analysis Form allowing users to input essential data and configure the analysis process.

To initiate the analysis, users must provide a valid, public GitHub repository URL. For a more targeted analysis, users can optionally set the From Date and To Date fields to define a custom time range. In case these fields are left blank, the default date range is automatically set to cover the past year. In order to determine the type of analysis to perform, users must select between Hotspot Prioritization and Commit Analysis using the radio buttons. Once the desired options are chosen, users can initiate the analysis process by clicking the "Inspect" button.

Upon clicking the "Inspect" button, the page initiates the analysis, sending a request to the back-end and redirecting the user to the Results Page. While the analysis is underway, a loading bar and the elapsed time are displayed and when the process is completed the final outcomes will be displayed.



Repo URL: https://github.com/Android-Development-UoM/PhysioAssistant-FrontEnd

From Date: mm / dd / yyyy

To Date: mm / dd / yyyy

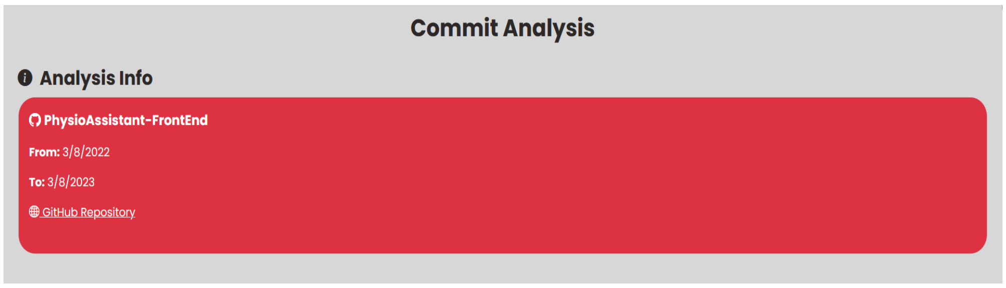
Figure 21 - CodeInspector Tool Page (Analysis Input Form)

In addition to the Analysis Form, the Tool Page provides convenient predefined analyses buttons for certain repositories. These buttons allow users to perform analysis tasks for repositories and date ranges without manually inputting data.

3.2.3 Results Page (Commit Analysis)

The Commit Analysis Page provides an overview of the repository's commit history, offering insights into the evolution of code changes over time. The Analysis Info section [Figure 16] displays essential project details, including the project's name, date range covered by the analysis, and a link to the corresponding GitHub repository for quick reference. The page serves as an essential tool for developers and other project stakeholders who wish to monitor a project's history.

Within the Commit Analysis Page, one of the components that enhance the understanding of the project's development lifecycle is the “Analysis Info Section”. This section (Figure 16), contains vital information about the analysis such as the repository url, the analysis date range and the analyzed project’s name. Understanding the analysis' time frame is critical for contextualizing code changes. For example, it enables users to spot activity spikes, periods of intense development, or periods of “quietness”. This data can help us understand a project's history, development speed, and the context in which specific code changes were performed.


Commit Analysis

Analysis Info

PhysioAssistant-FrontEnd

From: 3/8/2022

To: 3/8/2023

GitHub Repository

Figure 22 - Commit Analysis (Analysis Info)

The practice of evaluating the quality of code changes in a repository is critical for the ongoing maintenance and improvement of software development projects. To aid in this quest, the tool features a Bar Chart (Figure 17). This chart visually represents the distribution of “change ratings” for the repository's commits. By using this chart users can quickly understand the overall quality of code changes in the repository. The visual nature of the Bar Chart also makes it easier to spot outliers or anomalies. Users can quickly identify projects with exceptionally high or low commit ratings, which may need closer inspection. This feature is particularly useful for code review and quality assurance, as it allows for the prioritization of efforts to address code changes that may have a significant impact on the project's quality.

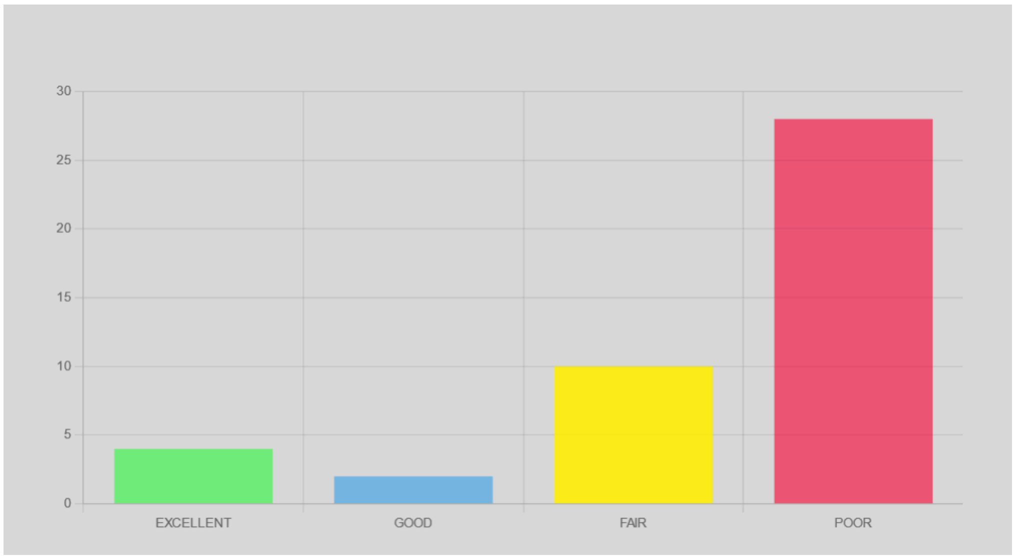
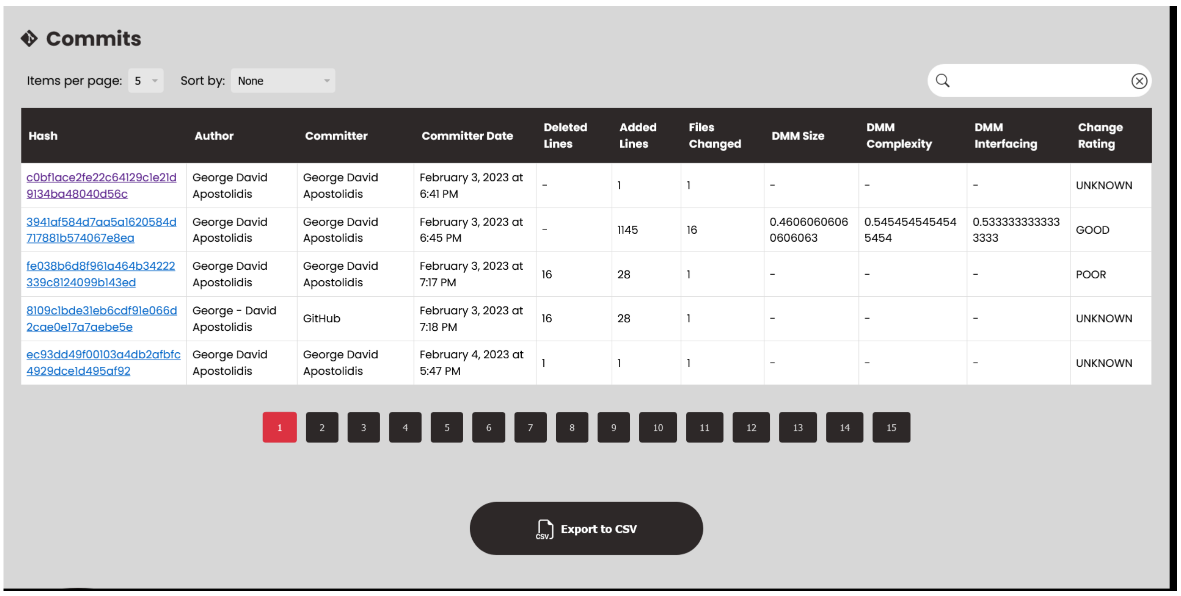


Figure 23 - Commits Bar Chart (Commits Analysis)

The Commits Table (Figure 18) is at the core of the page, with a dropdown menu for selecting columns to sort in ascending order and a search box for searching by author. Each row represents a distinct commit and includes relevant details such as the commit hash, author, date and more. The table is paginated for simple browsing, and a dropdown menu lets users choose how many entries are displayed on each page. To help users in further analysis, the page includes a "Export to CSV" button, allowing users to save the table into a file for additional analysis. Additionally users can click on the commit hash to view the individual commit on GitHub.



Hash	Author	Committer	Committer Date	Deleted Lines	Added Lines	Files Changed	DMM Size	DMM Complexity	DMM Interfacing	Change Rating
c0bf1ace2fe22c64129c1e21d9134ba48040d56c	George David Apostolidis	George David Apostolidis	February 3, 2023 at 6:41 PM	-	1	1	-	-	-	UNKNOWN
3941af584d7aa5a1620584d717881b574067e8ea	George David Apostolidis	George David Apostolidis	February 3, 2023 at 6:45 PM	-	1145	16	0.46060606060606063	0.5454545454545454	0.5333333333333333	GOOD
fe038b6d8f961a464b34222339c8124099b143ed	George David Apostolidis	George David Apostolidis	February 3, 2023 at 7:17 PM	16	28	1	-	-	-	POOR
8109c1bde31eb6cdf91e066d2cae0e17a7aebe5e	George - David Apostolidis	GitHub	February 3, 2023 at 7:18 PM	16	28	1	-	-	-	UNKNOWN
ec93dd49f00103a4db2afbfc4929dce1d495af92	George David Apostolidis	George David Apostolidis	February 4, 2023 at 5:47 PM	1	1	1	-	-	-	UNKNOWN

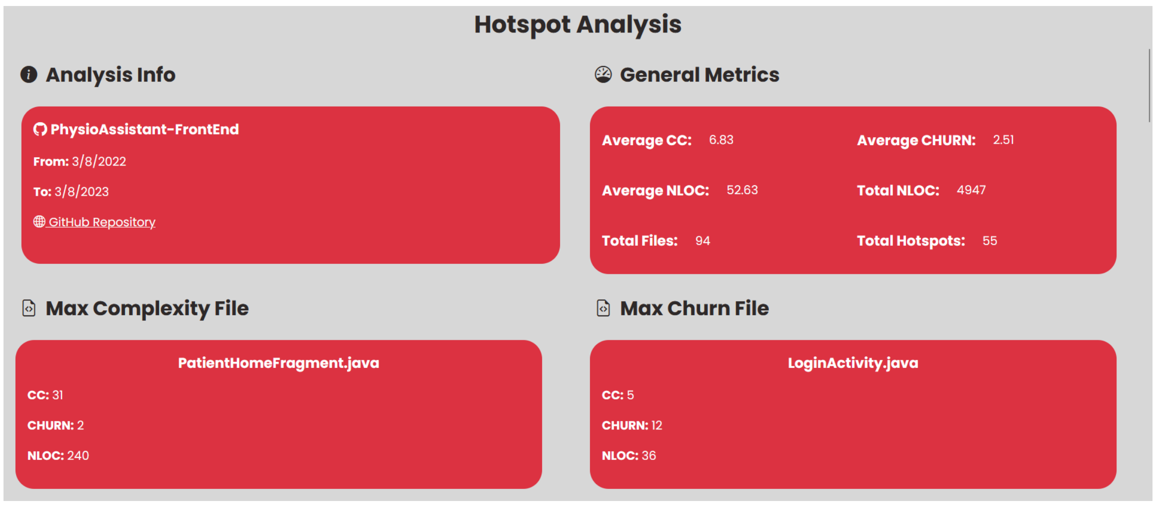
Figure 24 - Commit Analysis (Commits Table)

The Table provides an in-depth look at the repository's individual commits. It provides features that let users comprehend the evolution of the codebase and the contributions of specific authors. The table has a dropdown menu that allows users to select and sort columns in ascending order, making it easy to organize and study commit data based on specific constraints. This dynamic sorting capability is very useful when looking for patterns or tracking the progress of code changes in a repository. Furthermore, the Commits Table has a search field that allows users to easily filter contributions by author simplifying the process of isolating and analyzing specific team members' contributions. In addition to its sorting and searching capabilities, the table is created with user ease in mind including pagination, and allowing users to navigate through long commit histories by breaking them up into manageable chunks. Users can also modify their browsing experience by using a dropdown menu to change the number of entries displayed on each page.

The page's commitment to user-friendly navigation is reinforced by the ability to click on the commit hash within the table and seamlessly navigate to the individual commit's GitHub page. All these small features ensure that users can efficiently study and analyze the commit history at their own pace. The table's versatility and accessibility make it a vital tool for developers, project managers, and anybody else involved in the software development process.

3.2.4 Results Page (Hotspot Analysis)

As with the Commit Analysis Page, the Hotspot Prioritization Analysis Page includes the Analysis Info section. The General Metrics section presents metrics that provide a comprehensive view of the repository's overall health and complexity. These metrics include *Average Cyclomatic Complexity*, *Average Churn*, *Average NLOC per file*, *Total NLOC*, *Total Files*, and *Total Hotspots*. Highlighting areas that require immediate attention, the *Max Complexity and Max Churn Files* sections showcase the files with the highest Cyclomatic Complexity (CC) and Churn metrics, along with their respective details such as Name, CC, Churn, and NLOC. Developers can use these info to identify files that might pose challenges in terms of maintainability and complexity.



Hotspot Analysis

Analysis Info

PhysioAssistant-FrontEnd
From: 3/8/2022
To: 3/8/2023
GitHub Repository

General Metrics

Average CC: 6.83
Average CHURN: 2.51
Average NLOC: 52.63
Total NLOC: 4947
Total Files: 94
Total Hotspots: 55

Max Complexity File

PatientHomeFragment.java
CC: 31
CHURN: 2
NLOC: 240

Max Churn File

LoginActivity.java
CC: 5
CHURN: 12
NLOC: 36

Figure 25 - Hotspot Analysis (General Info)

A visual representation of hotspots, the Hotspot Prioritization Matrix provides a scatter plot based on churn and CC metrics. Each point on the plot represents a file, and its color indicates the file's priority. Hovering over a point reveals additional details, including File Name, Cyclomatic Complexity, Churn, and NLOC. This matrix enables developers to pinpoint critical areas and gain a deeper understanding of the repository's overall quality.

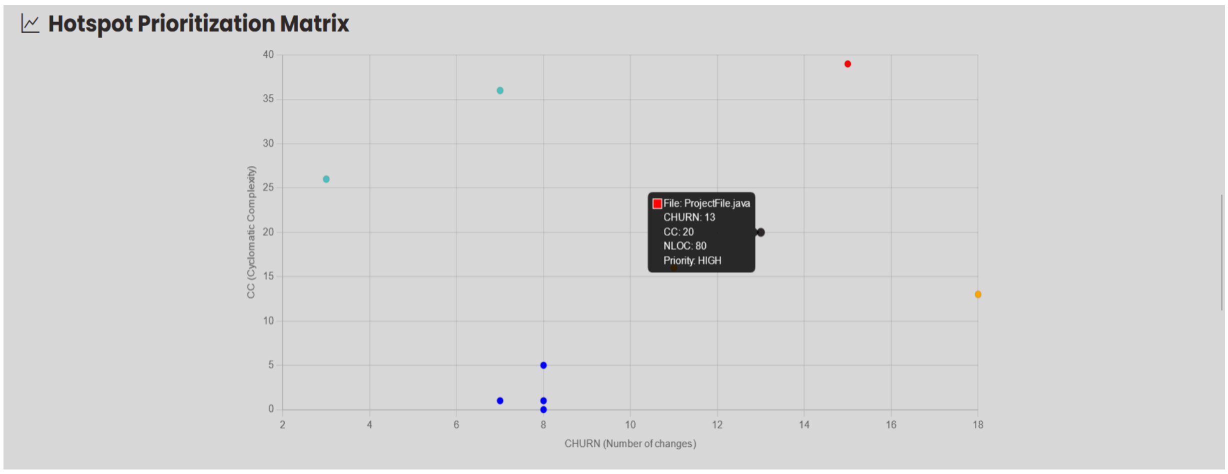
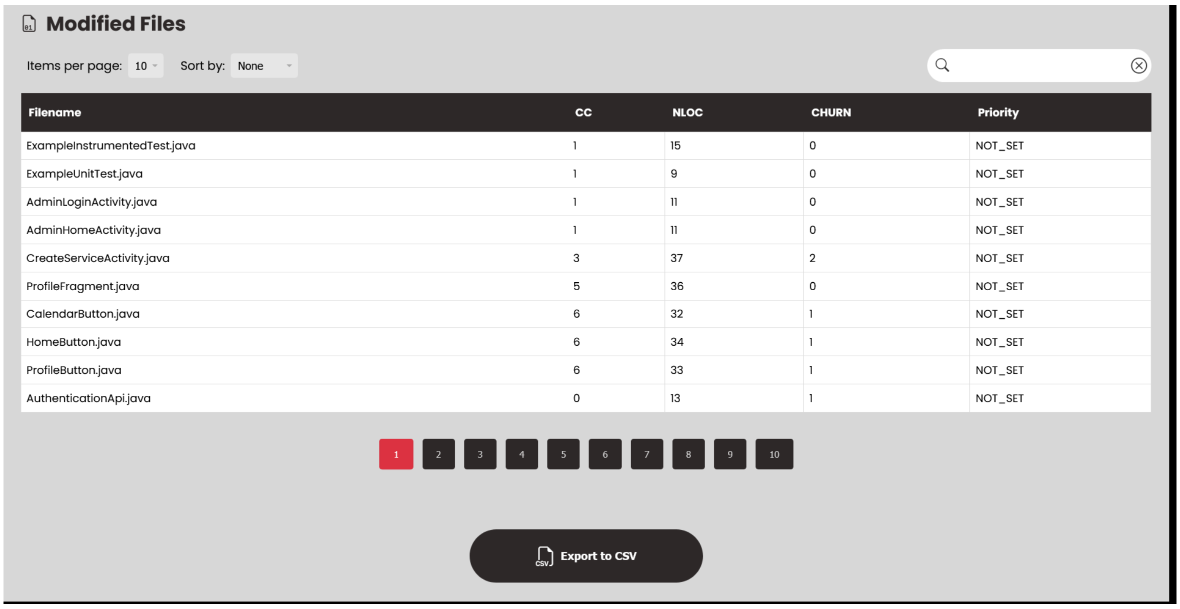


Figure 26 - Hotspot Analysis (Prioritization Matrix)

The Modified Files Table lists the repository's modified files, providing essential details such as Filename, Cyclomatic Complexity (CC), Number of Lines of Code (NLOC), Churn, and Priority. Like the Commits Table the Hotspots Table is paginated, searchable, and a dropdown menu lets users choose how many entries are displayed on each page. An "Export to CSV" button enables users to export the table data for further analysis.



Filename	CC	NLOC	CHURN	Priority
ExampleInstrumentedTest.java	1	15	0	NOT_SET
ExampleUnitTest.java	1	9	0	NOT_SET
AdminLoginActivity.java	1	11	0	NOT_SET
AdminHomeActivity.java	1	11	0	NOT_SET
CreateServiceActivity.java	3	37	2	NOT_SET
ProfileFragment.java	5	36	0	NOT_SET
CalendarButton.java	6	32	1	NOT_SET
HomeButton.java	6	34	1	NOT_SET
ProfileButton.java	6	33	1	NOT_SET
AuthenticationApi.java	0	13	1	NOT_SET

Figure 27 - Hotspot Analysis (Modified Files)

4 Tool Architecture

CodeInspector follows a client-server architecture, where the Frontend acts as the client, and the Backend acts as the server. The Front End of the tool provides a user-friendly interface that directly interacts with the Backend through a RESTful API, enabling efficient communication and data exchange between the user interface and the core processing engine. This client-server model enhances CodeInspector's usability and ensures that users can effortlessly access and leverage its analytical capabilities.

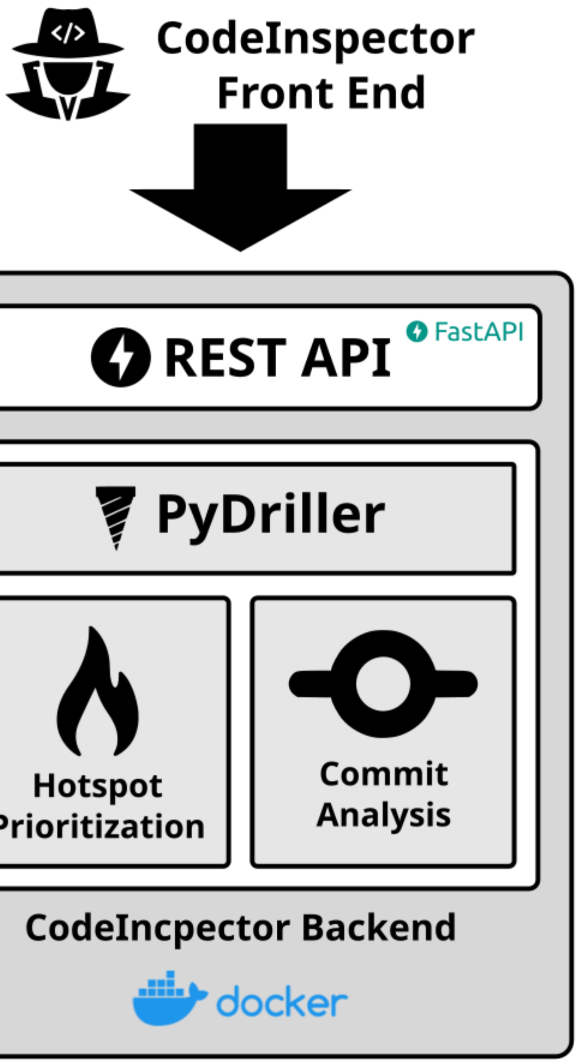


Figure 28 - CodeInspector Overall Architecture

4.1 Source Code Structure

The project's source code is divided into two main folders: “frontend” and “backend”, which are both hosted in the same GitHub repository (CodeInspector Repository). This structure promotes a unified approach to the project’s codebase management. The rationale behind this is to maintain a clear separation of concerns, making it more manageable and comprehensible for contributors, users, and developers.

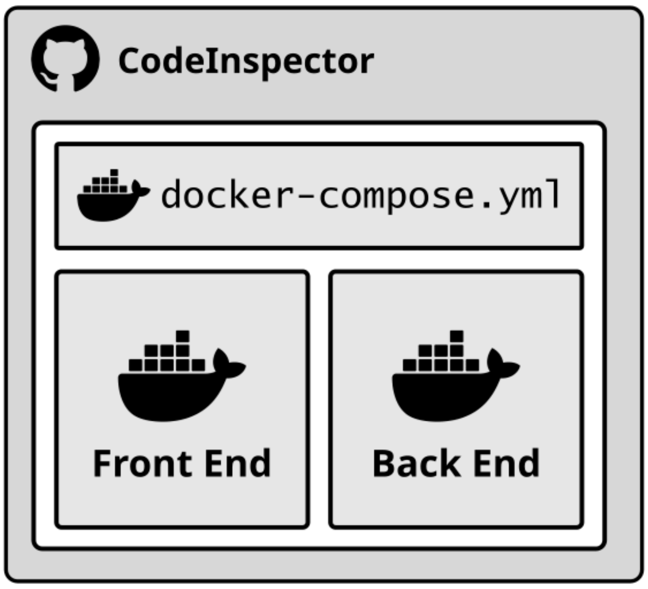
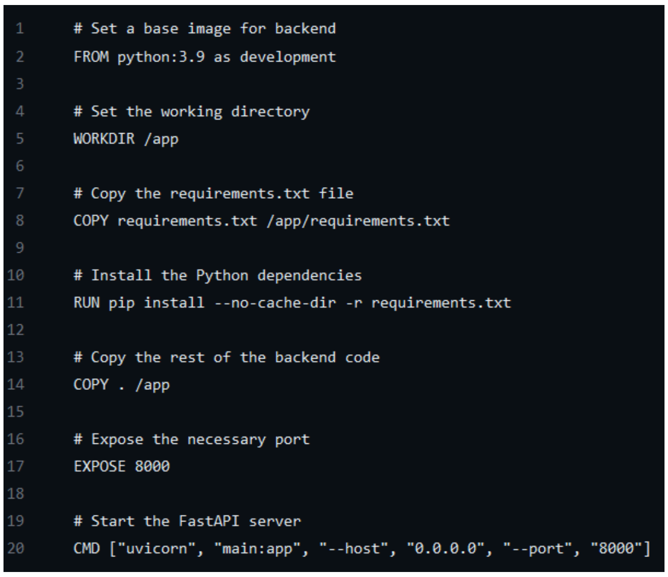


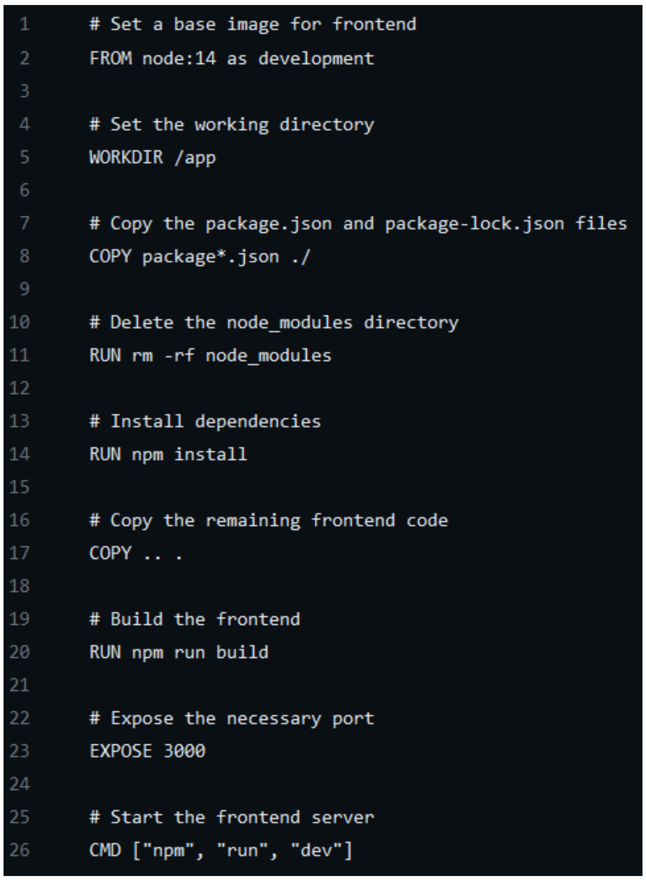
Figure 29 - CodeInspector Repository Structure

The backend folder is home to all the server-side logic and functionality as well as the `Dockerfile` (Code Block 1) that specifies how the backend component should be encapsulated in a container. The frontend folder contains the client-side code and assets responsible for the user interface and experience. Just like the backend, this folder also includes a `Dockerfile` (Code Block 2) that dictates how the component is containerized.



```
# Set a base image for backend
FROM python:3.9 as development

# Set the working directory
WORKDIR /app

# Copy the requirements.txt file
COPY requirements.txt /app/requirements.txt

# Install the Python dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of the backend code
COPY . /app

# Expose the necessary port
EXPOSE 8000

# Start the FastAPI server
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

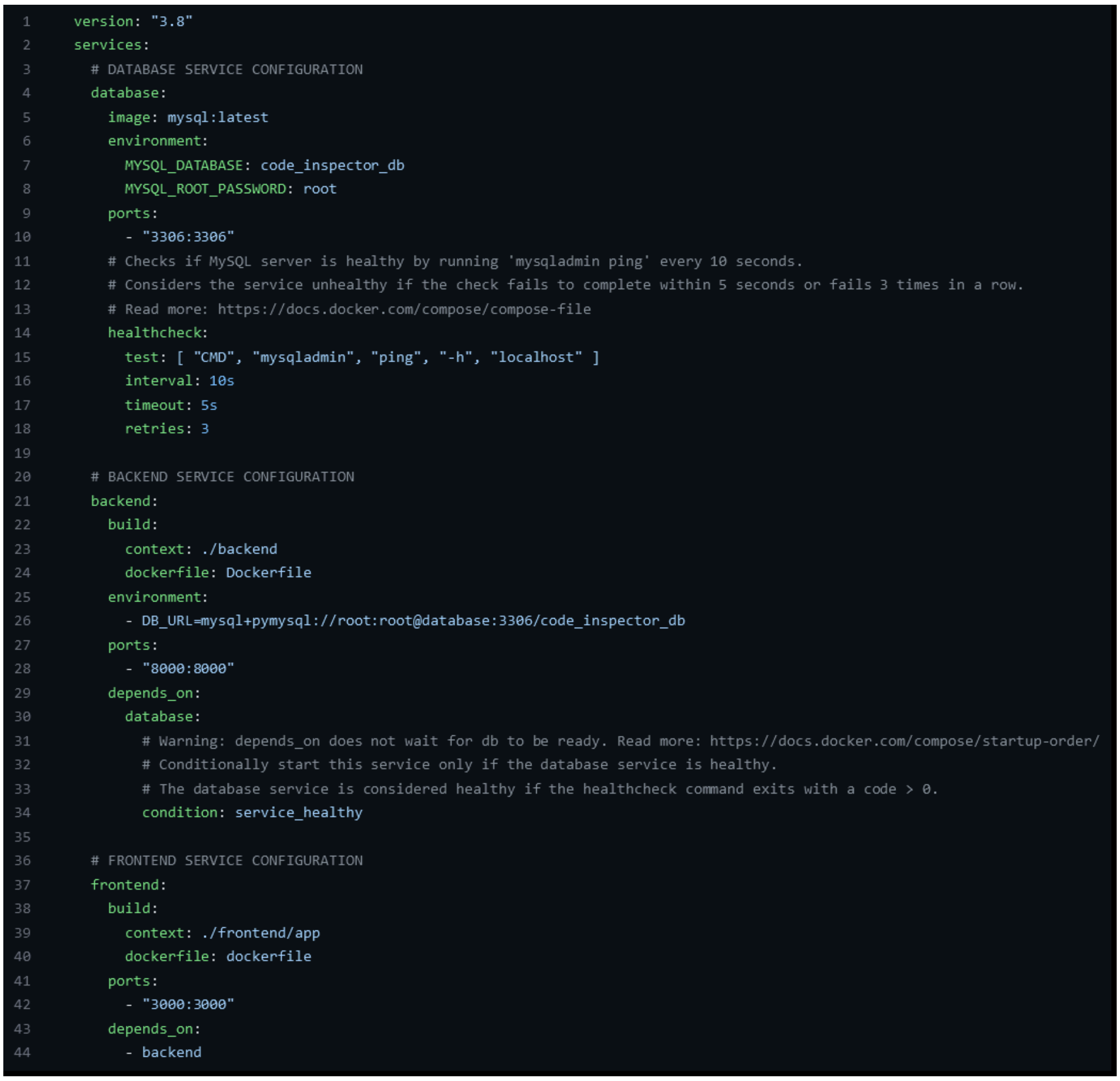
Code Block 1 - Back End Dockerfile



```
# Set a base image for frontend
FROM node:14 as development

# Set the working directory
WORKDIR /app

# Copy the package.json and package-lock.json files
COPY package*.json ./

# Delete the node_modules directory
RUN rm -rf node_modules

# Install dependencies
RUN npm install

# Copy the remaining frontend code
COPY .. .

# Build the frontend
RUN npm run build

# Expose the necessary port
EXPOSE 3000

# Start the frontend server
CMD ["npm", "run", "dev"]
```

Code Block 2 - Front End Dockerfile

To bring these components together and facilitate the deployment of the full-stack application, the project includes a `docker-compose.yml` file. This file defines the services required for the application and configures how they should interact, orchestrating the deployment and management of both frontend, backend and database containers.



```yaml
version: "3.8"
services:
  # DATABASE SERVICE CONFIGURATION
  database:
    image: mysql:latest
    environment:
      MYSQL_DATABASE: code_inspector_db
      MYSQL_ROOT_PASSWORD: root
    ports:
      - "3306:3306"
    # Checks if MySQL server is healthy by running 'mysqladmin ping' every 10 seconds.
    # Considers the service unhealthy if the check fails to complete within 5 seconds or fails 3 times in a row.
    # Read more: https://docs.docker.com/compose/compose-file
    healthcheck:
      test: [ "CMD", "mysqladmin", "ping", "-h", "localhost" ]
      interval: 10s
      timeout: 5s
      retries: 3

  # BACKEND SERVICE CONFIGURATION
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    environment:
      - DB_URL=mysql+pymysql://root:root@database:3306/code_inspector_db
    ports:
      - "8000:8000"
    depends_on:
      database:
        # Warning: depends_on does not wait for db to be ready. Read more: https://docs.docker.com/compose/startup-order/
        # Conditionally start this service only if the database service is healthy.
        # The database service is considered healthy if the healthcheck comand exits with a code > 0.
        condition: service_healthy

  # FRONTEND SERVICE CONFIGURATION
  frontend:
    build:
      context: ./frontend/app
      dockerfile: dockerfile
    ports:
      - "3000:3000"
    depends_on:
      - backend
```

Code Block 3 - docker-compose.yml

4.2 Backend Structure

The tool's backend is organized into different directories, each serving a specific purpose. This structure helps to ensure a clean and maintainable codebase.

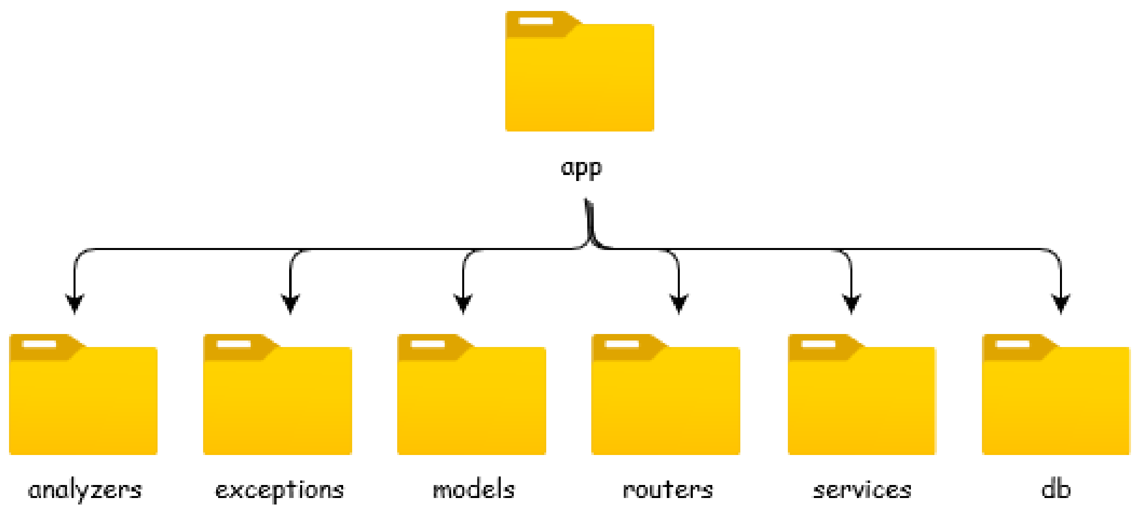


Figure 30 - CodeInspector Backend Structure

On the above figure, we can see the representation of the file structure for the backend. Let's explore each directory and its role:

1. **app**

   This is the top-level directory. It acts as the root of the backend containing all the core functionalities. It serves as a container for all other modules and packages, providing a central location to manage the application.

2. **analyzers**

   This directory hosts modules responsible for analyzing code and calculating various metrics. These modules collaborate to analyze projects.

3. **exceptions**

   This directory contains custom exception classes. It includes `InvalidDateException` and `NoCommitsException`, which are raised when encountering invalid dates or no commits during analysis.

4. **models**

   This directory defines data the models used throughout the analysis processes. Classes such as `Analysis`, `CommitAnalysis`, `PriorityAnalysis`, `ProjectCommit`, and `ProjectCommitBuilder` are present here.

5. **routers**

   This directory contains modules defining the API routes and endpoints. It includes the `analysis_router.py` module, which defines routes for hotspot prioritization and commit analysis. These routes serve as endpoints that trigger the respective analysis tasks when accessed by users through the frontend.

6. **services**

   The services directory contains the core service module AnalysisService. The module coordinates the entire analysis process, including both hotspot prioritization and commit analysis.

7. **db**

   This directory contains the database schemas and configuration modules. The directory is responsible for the database configuration and holds the models related to the database tables.

4.3 Hotspot Prioritization Analysis Implementation

The hotspot prioritization analysis in Code Inspector is responsible for identifying and prioritizing hotspot files in the codebase based on their Cyclomatic Complexity (CC) and Churn metrics. We consider a file as a Hotspot if it has higher CC or Churn compared to the average values of the project. The workflow of the analysis is shown on the following diagram:

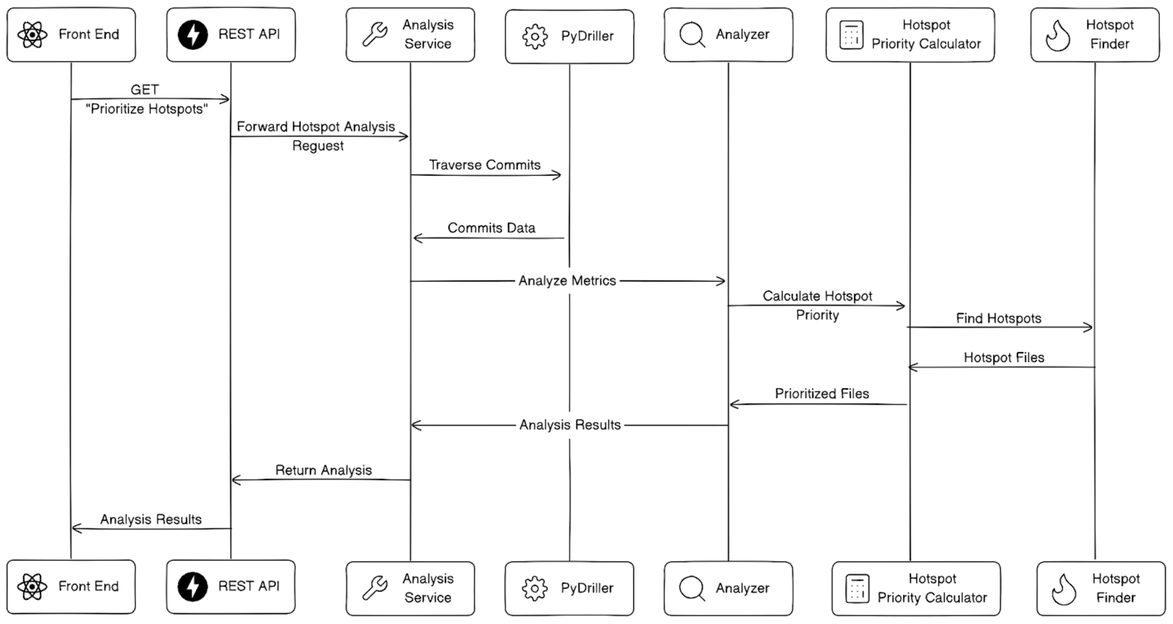
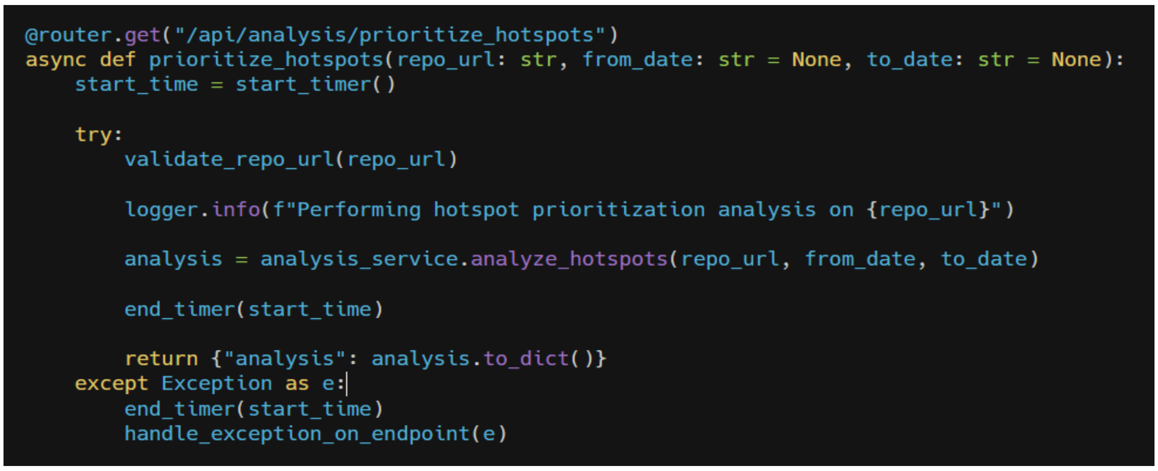


Figure 31 - Hotspot Analysis Workflow

4.3.1 API Endpoint (Hotspot Prioritization)

The hotspot prioritization analysis is triggered by an API endpoint. When a user initiates this process by making a request to the `/api/analysis/prioritize_hotspots` endpoint, it sets in motion a series of operations within the backend of our system, driven by the `AnalysisService` class. When a request is made to this endpoint, the provided repository URL is subjected to a validation process to ensure it is a valid github url and once the validation is successfully completed, the system logs a message, indicating that the analysis process is initiated.

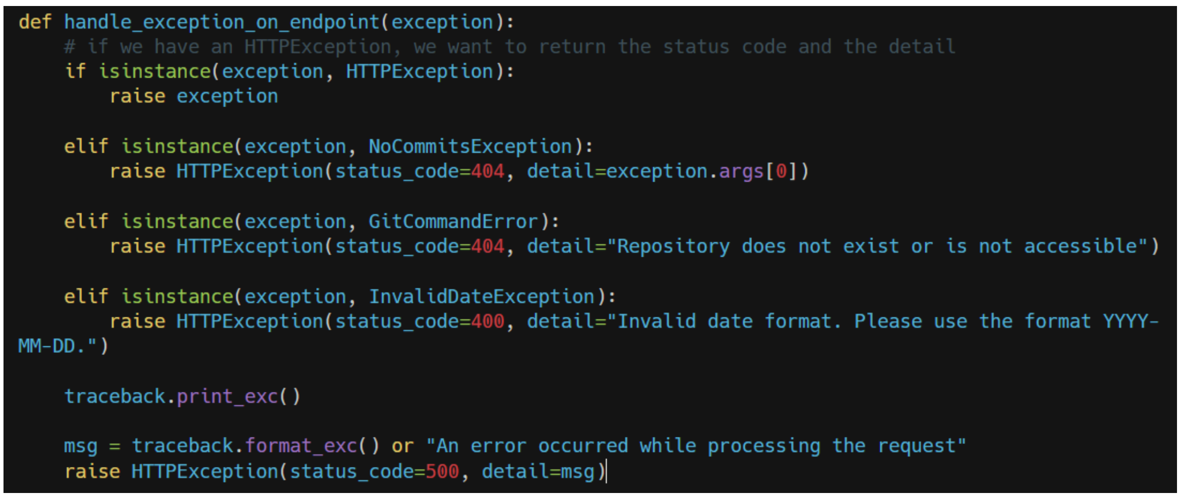


```python
@router.get("/api/analysis/prioritize_hotspots")
async def prioritize_hotspots(repo_url: str, from_date: str = None, to_date: str = None):
    start_time = start_timer()

    try:
        validate_repo_url(repo_url)

        logger.info(f"Performing hotspot prioritization analysis on {repo_url}")

        analysis = analysis_service.analyze_hotspots(repo_url, from_date, to_date)

        end_timer(start_time)

        return {"analysis": analysis.to_dict()}
    except Exception as e:
        end_timer(start_time)
        handle_exception_on_endpoint(e)
```

Code Block 4 - Prioritize Hotspots Endpoint (analysis_routers.py)

The center of the analysis process lies within the `analyze_hotspots` method of the `AnalysisService` Class. The result of this analysis is encapsulated within the 'analysis' variable. The system records the starting time using the `start_timer()` function at the beginning of the analysis and ends it with the `end_timer()` function once the analysis is complete. This allows for precise measurement and monitoring of the analysis duration. Upon successful analysis, the results are returned to the user as a JSON response, with the analysis data encapsulated in the 'analysis' field. This data can be conveniently converted to a dictionary, making it accessible and consumable for further processing or presentation.

In the event of an error during any part of this process, the system is equipped to handle and manage it using the `handle_exception_on_endpoint()` function. This ensures that the application remains robust and resilient, even in the face of unexpected issues.

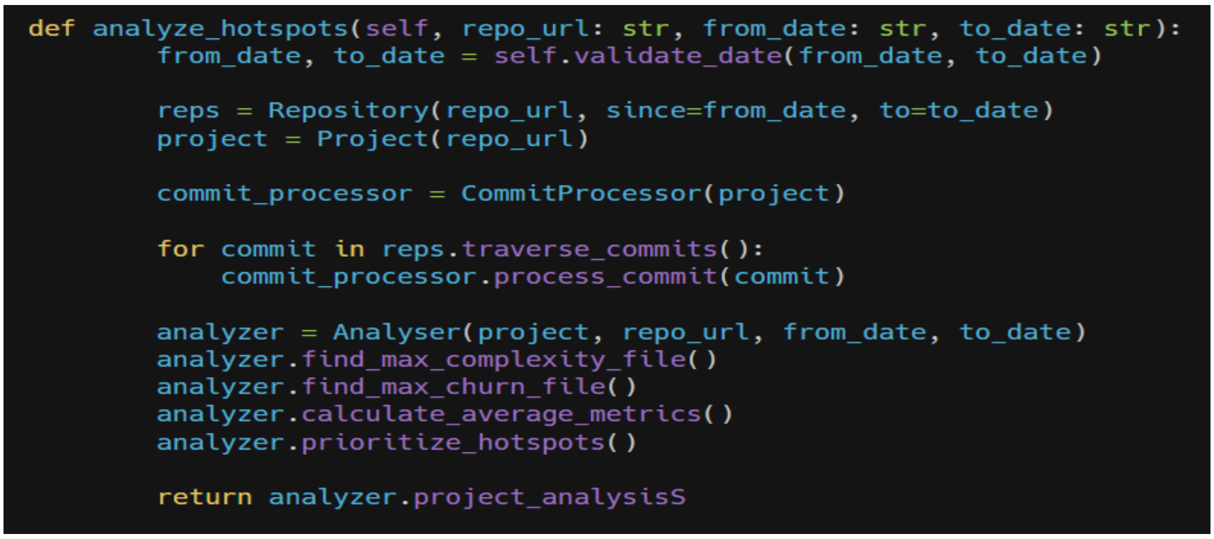


```
def handle_exception_on_endpoint(exception):
    # if we have an HTTPException, we want to return the status code and the detail
    if isinstance(exception, HTTPException):
        raise exception

    elif isinstance(exception, NoCommitsException):
        raise HTTPException(status_code=404, detail=exception.args[0])

    elif isinstance(exception, GitCommandError):
        raise HTTPException(status_code=404, detail="Repository does not exist or is not accessible")

    elif isinstance(exception, InvalidDateException):
        raise HTTPException(status_code=400, detail="Invalid date format. Please use the format YYYY-
MM-DD.")

    traceback.print_exc()

    msg = traceback.format_exc() or "An error occurred while processing the request"
    raise HTTPException(status_code=500, detail=msg)
```

Code Block 5 - Exception Handling Method (__init__.py | 'routers' module)

4.3.2 The AnalysisService Class (Hotspot Prioritization)

The AnalysisService class has an "`analyze_hotspots`" method that is responsible for coordinating the hotspot prioritization analysis. It initializes the analysis with the necessary parameters, such as the repository URL and date range. It uses the PyDriller library classes to mine essential data from the specified repository and the Analyzer class to analyze the received data.



```
def analyze_hotspots(self, repo_url: str, from_date: str, to_date: str):
        from_date, to_date = self.validate_date(from_date, to_date)

        reps = Repository(repo_url, since=from_date, to=to_date)
        project = Project(repo_url)

        commit_processor = CommitProcessor(project)

        for commit in reps.traverse_commits():
            commit_processor.process_commit(commit)

        analyzer = Analyser(project, repo_url, from_date, to_date)
        analyzer.find_max_complexity_file()
        analyzer.find_max_churn_file()
        analyzer.calculate_average_metrics()
        analyzer.prioritize_hotspots()

        return analyzer.project_analysisS
```

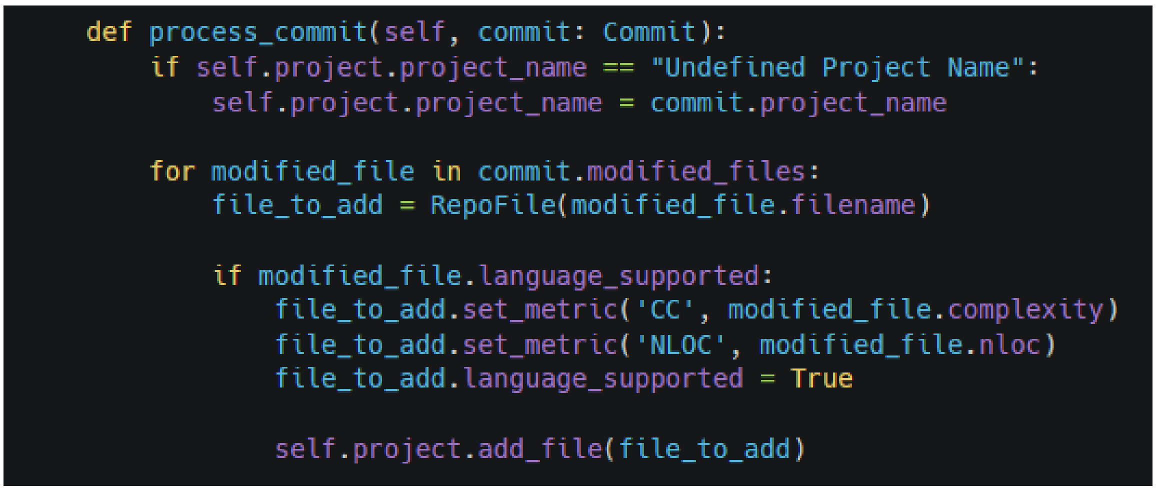
Code Block 6 - Analyze Hotspots Method (AnalyzerService.py)

The method begins by validating the provided date range using the `validate_date` method of the current class. Then initializes a PyDriller `Repository` object with the specified url and date range. Next, it creates a `Project` object and sets up a

`CommitProcessor` to process each commit. Then an Analyzer object is created to analyze the project's data and prioritize hotspots. When the analysis is completed, the method returns the results.

4.3.3 The CommitProcessor Class (Hotspot Prioritization)

This class analyzes and processes commits from a GitHub repository. It takes a project object as input, which holds information about the project being analyzed. It contains one method responsible for processing each commit in the repository.



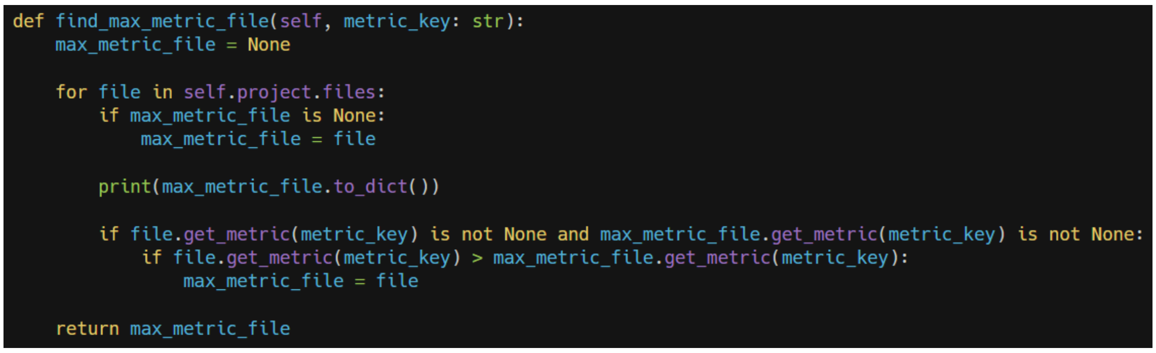
```python
def process_commit(self, commit: Commit):
    if self.project.project_name == "Undefined Project Name":
        self.project.project_name = commit.project_name

    for modified_file in commit.modified_files:
        file_to_add = RepoFile(modified_file.filename)

        if modified_file.language_supported:
            file_to_add.set_metric('CC', modified_file.complexity)
            file_to_add.set_metric('NLOC', modified_file.nloc)
            file_to_add.language_supported = True

            self.project.add_file(file_to_add)
```

Code Block 7 - Process Commit Method (CommitProcessor.py)

The method takes a commit object from the PyDriller library. The `Commit` class represents a single commit in the repository and contains various data, such as modified files, commit message, committer, and more. Upon processing a commit, the class checks if the project's name is set. If it is not, the method sets the project name based on the commit project name. Next, the method iterates through the modified files in the commit and for each file, a new `RepoFile` object is created. `RepoFile` is a model class that represents a file in the repository. Before adding the file to the project, the method checks if the file's language is supported. If the language is supported, the method sets the file's metrics. After this, the language_supported attribute of the RepoFile is set to True, indicating that the file's language is supported and the metrics are available. Finally, the processed RepoFile object is added to the project.

4.3.4 The Analyzer Class (Hotspot Prioritization)

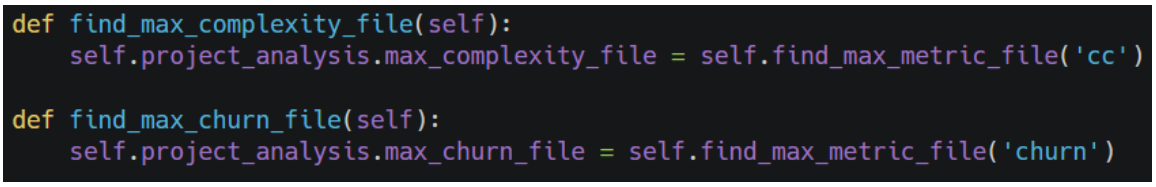
This class is a central part of the hotspot prioritization analysis. It receives the Project object, which contains information about the modified files in the repository. The class performs several essential tasks, including finding the files with the highest CC and Churn metrics and prioritizing hotspots based on these metrics. Let's examine each method in detail:



```python
def find_max_metric_file(self, metric_key: str):
    max_metric_file = None

    for file in self.project.files:
        if max_metric_file is None:
            max_metric_file = file

        print(max_metric_file.to_dict())

        if file.get_metric(metric_key) is not None and max_metric_file.get_metric(metric_key) is not None:
            if file.get_metric(metric_key) > max_metric_file.get_metric(metric_key):
                max_metric_file = file

    return max_metric_file
```

Code Block 8 - Find Max Metric File Method (Analyzer.py)

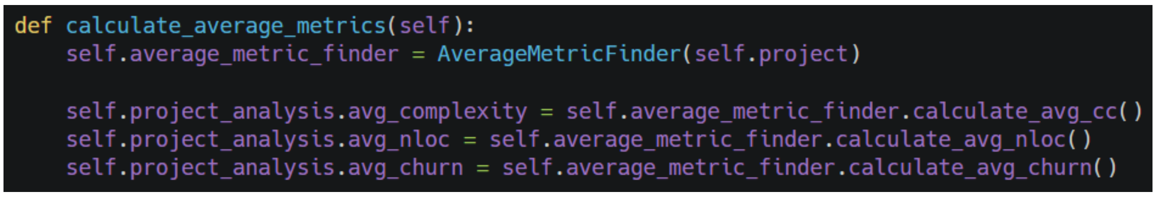
The `find_max_metric_file` method is responsible for finding the file with the highest value for a specified metric (CC or Churn) in the project. The method iterates through all the files in the project, compares their metric values with the current maximum value, and updates the max_metric_file variable accordingly. At the end of the iteration, the method returns the file with the highest metric value.



```python
def find_max_complexity_file(self):
    self.project_analysis.max_complexity_file = self.find_max_metric_file('cc')

def find_max_churn_file(self):
    self.project_analysis.max_churn_file = self.find_max_metric_file('churn')
```

Code Block 9 - Find Max Metric File Method (Analyzer.py)

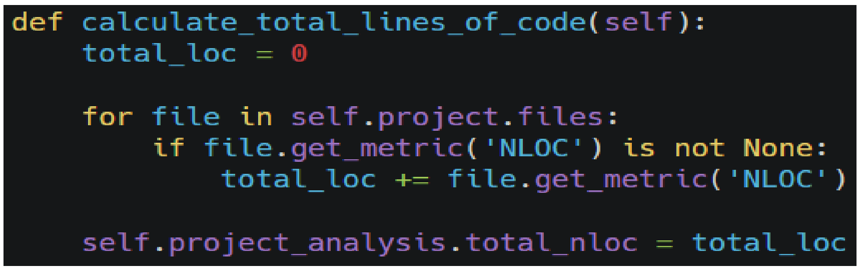
These two methods are responsible for identifying the files with the highest Cyclomatic Complexity (CC) and Churn metrics in the project, respectively. The methods set the max_complexity_file or max_churn_file attribute of the project_analysis object by calling the `find_max_metric_file` method.



```python
def calculate_average_metrics(self):
    self.average_metric_finder = AverageMetricFinder(self.project)

    self.project_analysis.avg_complexity = self.average_metric_finder.calculate_avg_cc()
    self.project_analysis.avg_nloc = self.average_metric_finder.calculate_avg_nloc()
    self.project_analysis.avg_churn = self.average_metric_finder.calculate_avg_churn()
```

Code Block 10 - Calculate Average Metrics Method (Analyzer.py)

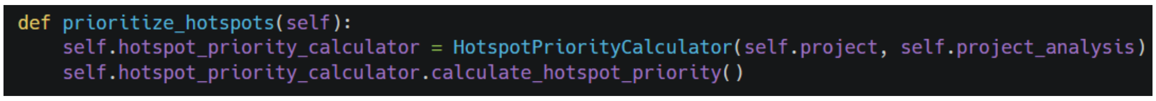
This method is used to calculate the average values of CC, NLOC, and Churn for the project. The method creates an AverageMetricFinder object that performs the calculation process.



```python
def calculate_total_lines_of_code(self):
    total_loc = 0

    for file in self.project.files:
        if file.get_metric('NLOC') is not None:
            total_loc += file.get_metric('NLOC')

    self.project_analysis.total_nloc = total_loc
```

Code Block 11 - Calculate Total LOC (Analyzer.py)

This method calculates the total number of lines of code (NLOC) in the project. It iterates through all the files and sums up their NLOC metric values, storing the result in the `total_nloc` attribute of the `project_analysis` object.



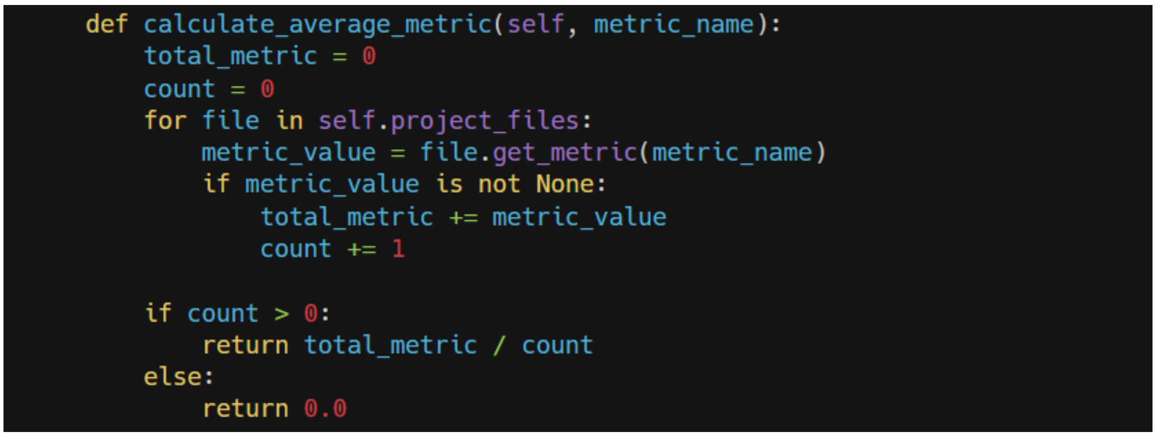
```python
def prioritize_hotspots(self):
    self.hotspot_priority_calculator = HotspotPriorityCalculator(self.project, self.project_analysis)
    self.hotspot_priority_calculator.calculate_hotspot_priority()
```

Code Block 12 - Prioritize Hotspots Method (Analyzer.py)

This method initiates the hotspot prioritization calculation. The method creates an instance of the `HotspotPriorityCalculator` class, passing the required objects as parameters. Then it calls the `calculate_hotspot_priority` method of `HotspotPriorityCalculator` to perform the actual prioritization.

4.3.5 The AverageMetricFinder Class (Hotspot Prioritization)

The class is responsible for analyzing and calculating average metrics (Cyclomatic Complexity - CC, Number of Lines of Code - NLOC, and Code Churn - CHURN) for a given project. The class has four methods:



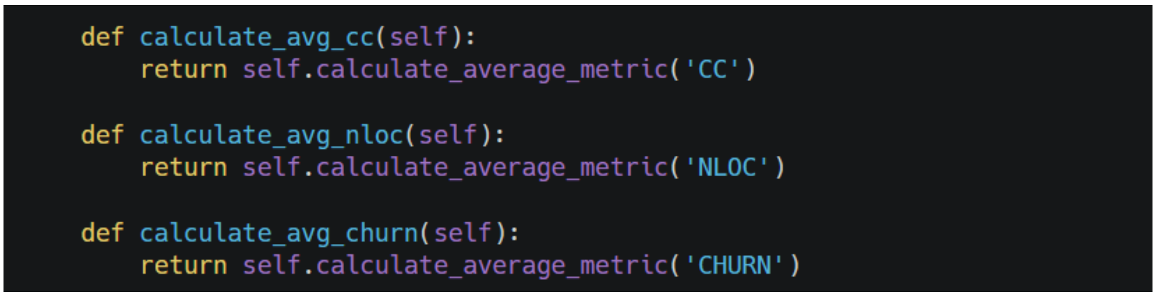
```
def calculate_average_metric(self, metric_name):
    total_metric = 0
    count = 0
    for file in self.project_files:
        metric_value = file.get_metric(metric_name)
        if metric_value is not None:
            total_metric += metric_value
            count += 1

    if count > 0:
        return total_metric / count
    else:
        return 0.0
```

Code Block 13 - Calculate Average Metric (AverageMetricFinder.py)

This method contains the core functionality of the class. The method takes the name of the metric to be calculated as an argument and calculates the average value (CC, NLOC, or CHURN) across all the modified files in the project.

The calculation is quite straightforward. The method initializes the `total_metric` and `count` variables to zero (0). Then the method iterates through the files in the project. For each file, it calls the `get_metric` method of the ProjectFile to retrieve the corresponding metric value. After verifying the metric value, it adds the value to the `total_metric` variable, and `count` is incremented by 1. The process continues until all the files have been processed.

After iterating through all files, the method calculates the average metric value by dividing total_metric by count. If there are no valid metric values (count is 0), the average is set to 0.0 to avoid division by zero errors. The method then returns the calculated value. The other three methods in the class are convenience methods that call the `calculate_average_metric` method with a specific metric name.

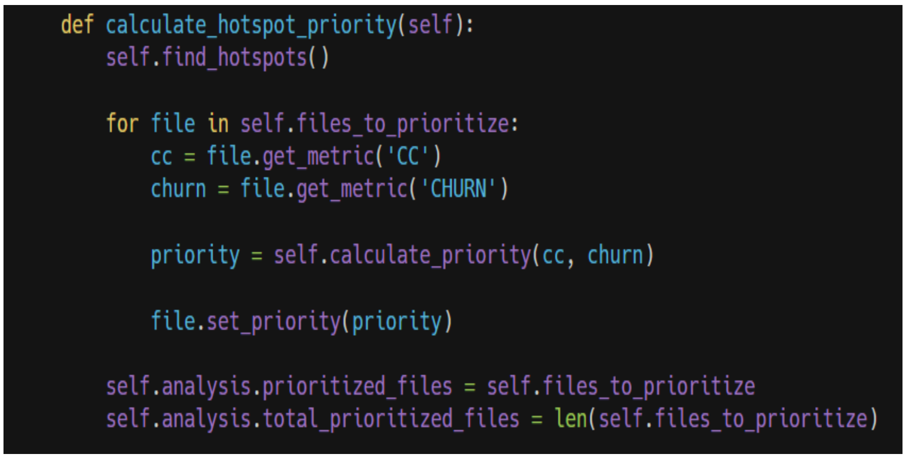


```python
    def calculate_avg_cc(self):
        return self.calculate_average_metric('CC')

    def calculate_avg_nloc(self):
        return self.calculate_average_metric('NLOC')

    def calculate_avg_churn(self):
        return self.calculate_average_metric('CHURN')
```

Code Block 14 - Convenience Methods (AverageMetricFinder.py)

4.3.6 The HotspotPriorityCalculator Class (Hotspot Prioritization)

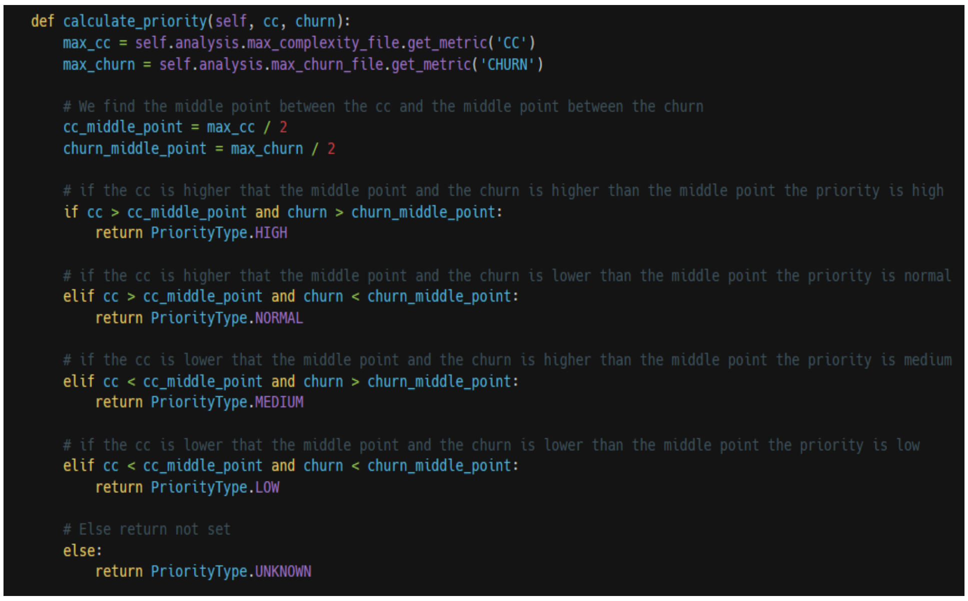
The `HotspotPriorityCalculator` class is responsible for calculating the priority of hotspot files based on their Cyclomatic Complexity and Churn metrics. The class collaborates with the `HotspotFinder` class, which identifies hotspots based on the project's average values of the metrics. Let's dive deeper into the implementation of the class.



```python
    def calculate_hotspot_priority(self):
        self.find_hotspots()

        for file in self.files_to_prioritize:
            cc = file.get_metric('CC')
            churn = file.get_metric('CHURN')

            priority = self.calculate_priority(cc, churn)

            file.set_priority(priority)

        self.analysis.prioritized_files = self.files_to_prioritize
        self.analysis.total_prioritized_files = len(self.files_to_prioritize)
```

Code Block 15 - Calculate Hotspot Priority (HotspotPriorityCalculator.py)

The above method is `calculate_hotspot_priority`, which is the main part of the class and triggers the entire prioritization process. It first calls the `find_hotspots` method to identify the hotspot files in the codebase. The identified files are then prioritized using the `calculate_priority` method, and their priority levels are set accordingly.

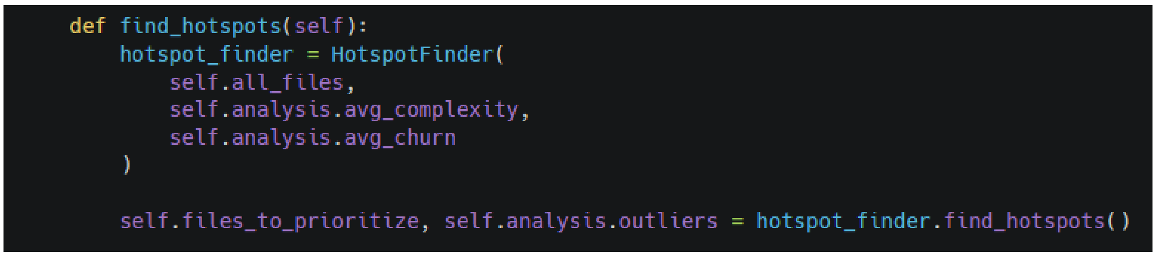


```python
def calculate_priority(self, cc, churn):
    max_cc = self.analysis.max_complexity_file.get_metric('CC')
    max_churn = self.analysis.max_churn_file.get_metric('CHURN')

    # We find the middle point between the cc and the middle point between the churn
    cc_middle_point = max_cc / 2
    churn_middle_point = max_churn / 2

    # if the cc is higher that the middle point and the churn is higher than the middle point the priority is high
    if cc > cc_middle_point and churn > churn_middle_point:
        return PriorityType.HIGH

    # if the cc is higher that the middle point and the churn is lower than the middle point the priority is normal
    elif cc > cc_middle_point and churn < churn_middle_point:
        return PriorityType.NORMAL

    # if the cc is lower that the middle point and the churn is higher than the middle point the priority is medium
    elif cc < cc_middle_point and churn > churn_middle_point:
        return PriorityType.MEDIUM

    # if the cc is lower that the middle point and the churn is lower than the middle point the priority is low
    elif cc < cc_middle_point and churn < churn_middle_point:
        return PriorityType.LOW

    # Else return not set
    else:
        return PriorityType.UNKNOWN
```

Code Block 16 - Calculate Priority Method (HotspotPriorityCalculator.py)

The `calculate_priority` method determines the priority of each hotspot by utilizing a technique similar to the Eisenhower Matrix based on their Cyclomatic Complexity and Churn metrics (for more details see Chapter 2.6). It calculates the middle points between the highest CC and Churn values and uses them to determine the priority of each file.

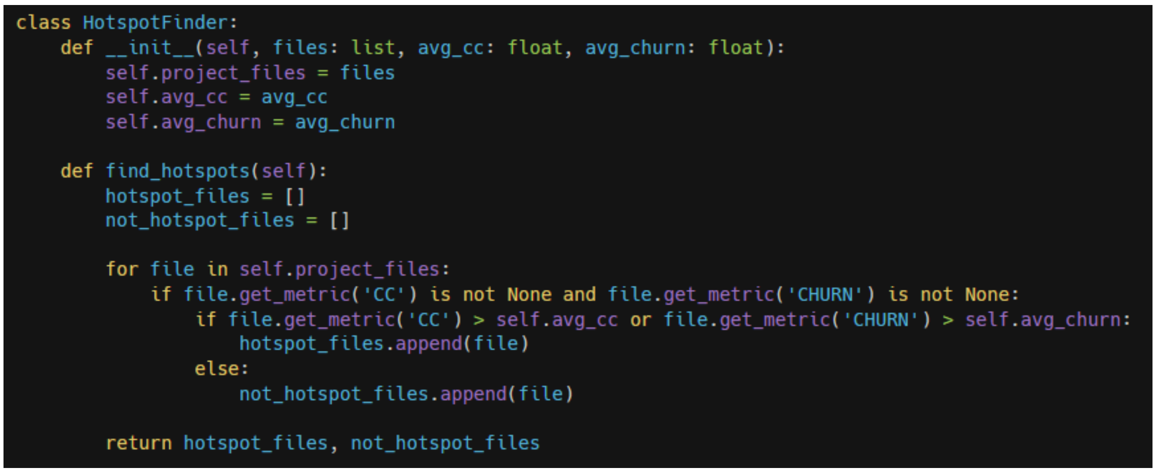
- If the CC and Churn values are both higher than their middle points, the file is assigned a high priority.
- If the CC is higher but the Churn is lower than their middle points, the file is assigned a normal priority.
- If the CC is lower but the Churn is higher than their middle points, the file is given a medium priority.
- If both CC and Churn are lower than their middle points, the file is assigned a low priority.
- If the file does not fall into any of these categories, it is labeled with an unknown (also referred as NOT SET) priority.



```python
    def find_hotspots(self):
        hotspot_finder = HotspotFinder(
            self.all_files,
            self.analysis.avg_complexity,
            self.analysis.avg_churn
        )

        self.files_to_prioritize, self.analysis.outliers = hotspot_finder.find_hotspots()
```

Code Block 17 - Calculate Priority Method (HotspotPriorityCalculator.py)

The `find_hotspots` method utilizes the `HotspotFinder` class to identify the hotspots based on the average CC and Churn of the entire project. It initializes a `HotspotFinder` object and passes it a of all files in the project and the average CC and Churn values. `HotspotFinder` then iterates through the files and checks if their CC and Churn values are higher than the corresponding average values. Based on this comparison, the method categorizes each file as a hotspot or an outlier.



```python
class HotspotFinder:
    def __init__(self, files: list, avg_cc: float, avg_churn: float):
        self.project_files = files
        self.avg_cc = avg_cc
        self.avg_churn = avg_churn

    def find_hotspots(self):
        hotspot_files = []
        not_hotspot_files = []

        for file in self.project_files:
            if file.get_metric('CC') is not None and file.get_metric('CHURN') is not None:
                if file.get_metric('CC') > self.avg_cc or file.get_metric('CHURN') > self.avg_churn:
                    hotspot_files.append(file)
                else:
                    not_hotspot_files.append(file)

        return hotspot_files, not_hotspot_files
```

Code Block 18 - HotspotFinder Class

4.4 Commit Analysis Implementation

The commit analysis in Code Inspector is responsible for analyzing and processing commits from a GitHub repository. The analysis is triggered by an API endpoint where a user makes a request to the `/api/analysis/commits` endpoint. The backend starts the analysis process using the `AnalysisService` class. The workflow of the analysis is shown on the following diagram:

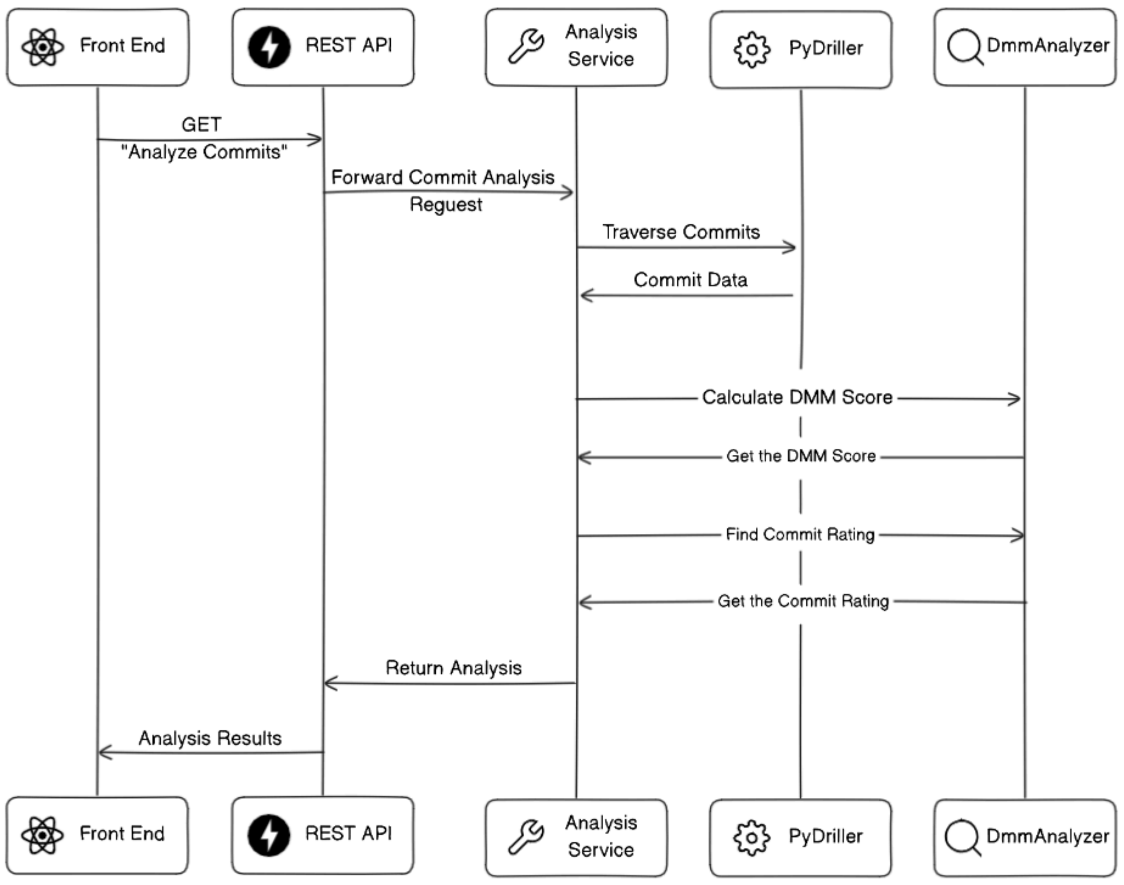
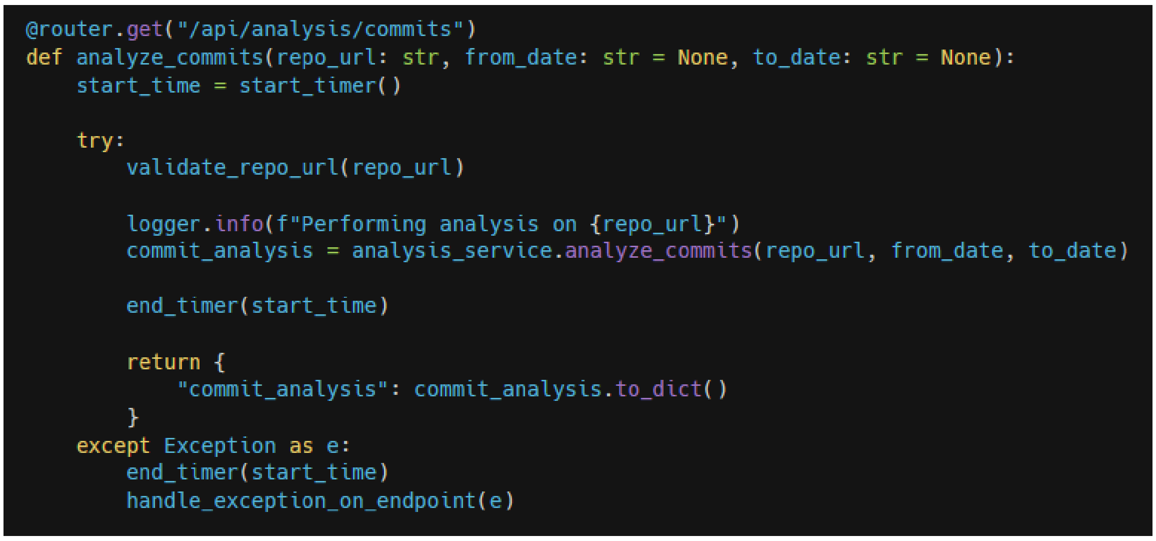


Figure 32 - Commit Analysis Workflow

4.4.1 API Endpoint (Hotspot Prioritization)

The API endpoint for commit analysis is also defined in `analysis_router.py`. When a GET request is made to this endpoint with the repository URL and optional date range, it validates the repository URL and initiates the commit analysis process. The result of the analysis is returned as a dictionary containing the commit analysis data.



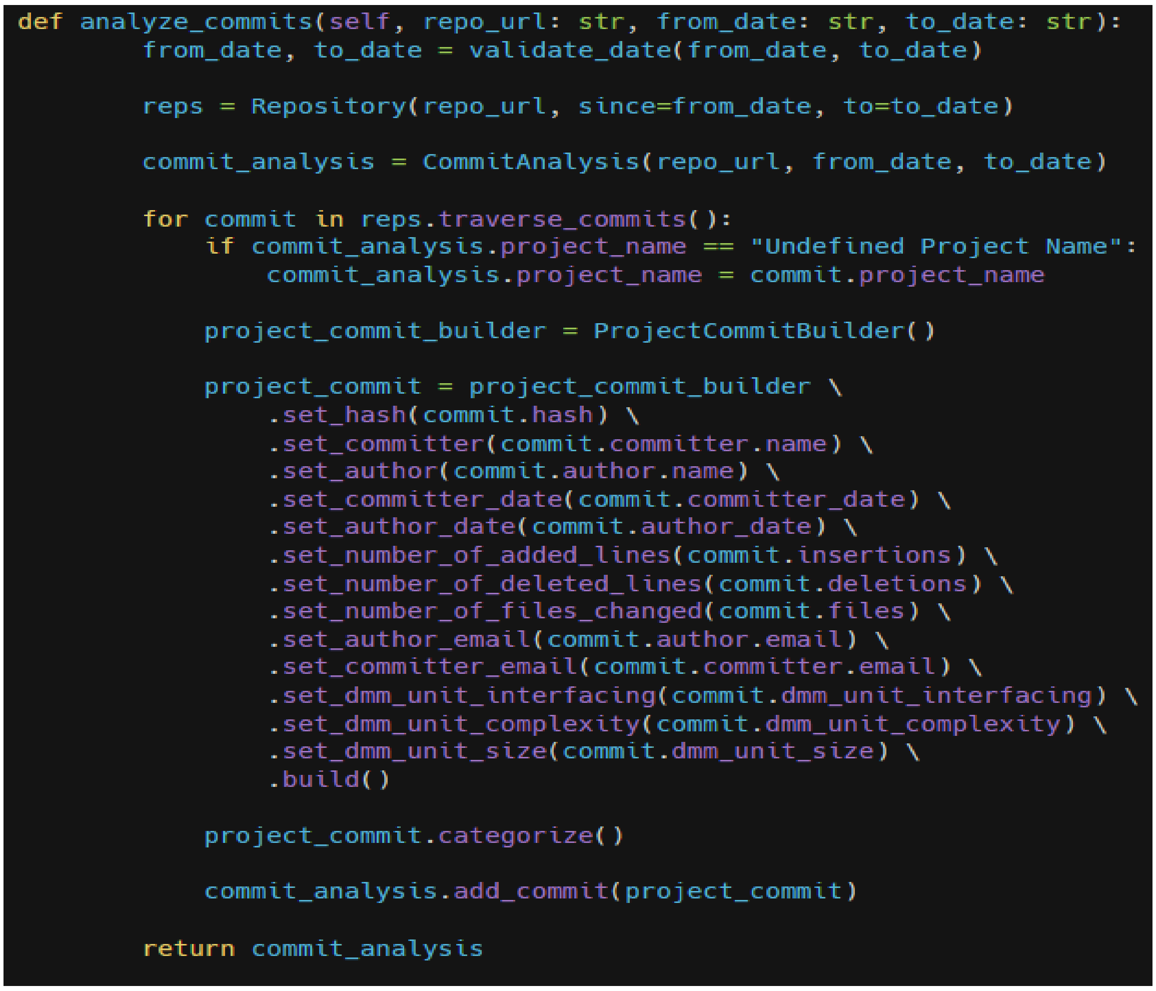
```python
@router.get("/api/analysis/commits")
def analyze_commits(repo_url: str, from_date: str = None, to_date: str = None):
    start_time = start_timer()

    try:
        validate_repo_url(repo_url)

        logger.info(f"Performing analysis on {repo_url}")
        commit_analysis = analysis_service.analyze_commits(repo_url, from_date, to_date)

        end_timer(start_time)

        return {
            "commit_analysis": commit_analysis.to_dict()
        }
    except Exception as e:
        end_timer(start_time)
        handle_exception_on_endpoint(e)
```

Code Block 19 - Commit Analysis Endpoint (analysis_routers.py)

4.4.2 The AnalysisService Class (Commit Analysis)

The `AnalysisService` class in `AnalysisService.py` has an `analyze_commits` method that coordinates the commit analysis. It initializes the analysis with necessary parameters such as the repository URL and date range. It uses the PyDriller library to mine essential data from the specified repository.

The method begins by validating the provided date range. Then, it initializes a PyDriller Repository object with the specified URL and date range. Next, it creates a `CommitAnalysis` object and processes each commit using a `ProjectCommitBuilder`. After the commit object is created, the AnalysisService class creates an instance of the DmmAnalyzer and calls the corresponding methods of the class to calculate the DMM Score and categorize the commit.



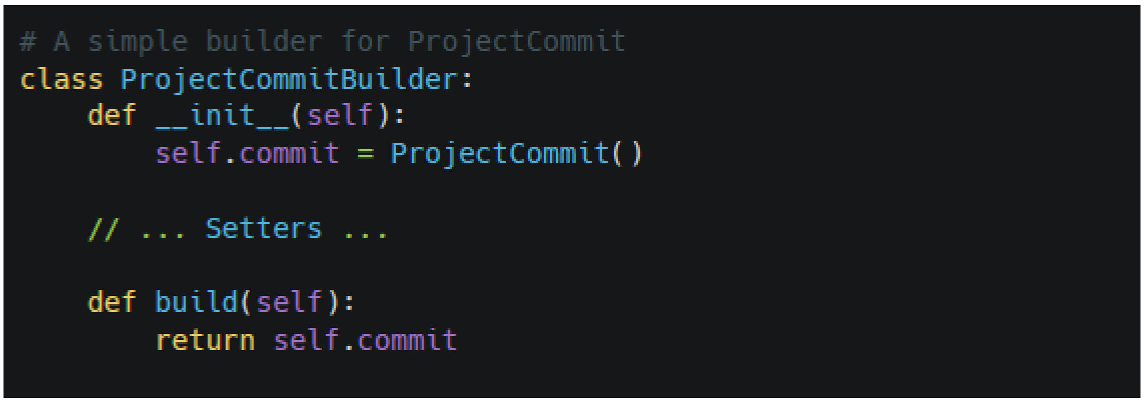
```python
def analyze_commits(self, repo_url: str, from_date: str, to_date: str):
        from_date, to_date = validate_date(from_date, to_date)

        reps = Repository(repo_url, since=from_date, to=to_date)

        commit_analysis = CommitAnalysis(repo_url, from_date, to_date)

        for commit in reps.traverse_commits():
            if commit_analysis.project_name == "Undefined Project Name":
                commit_analysis.project_name = commit.project_name

            project_commit_builder = ProjectCommitBuilder()

            project_commit = project_commit_builder \
                .set_hash(commit.hash) \
                .set_committer(commit.committer.name) \
                .set_author(commit.author.name) \
                .set_committer_date(commit.committer_date) \
                .set_author_date(commit.author_date) \
                .set_number_of_added_lines(commit.insertions) \
                .set_number_of_deleted_lines(commit.deletions) \
                .set_number_of_files_changed(commit.files) \
                .set_author_email(commit.author.email) \
                .set_committer_email(commit.committer.email) \
                .set_dmm_unit_interfacing(commit.dmm_unit_interfacing) \
                .set_dmm_unit_complexity(commit.dmm_unit_complexity) \
                .set_dmm_unit_size(commit.dmm_unit_size) \
                .build()

            project_commit.categorize()

            commit_analysis.add_commit(project_commit)

        return commit_analysis
```

Code Block 20 - Analyze Commits Method (AnalyzerService.py)

4.4.3 The ProjectCommitBuilder Class

The `ProjectCommitBuilder` class, defined in the `ProjectCommitBuilder.py` file, serves as a tool for constructing `ProjectCommit` instances with ease and precision. The class is an implementation of the builder pattern, making the construction of `ProjectCommit` objects straightforward and adaptable, ultimately enhancing the development experience. The `ProjectCommitBuilder` class has methods to set various attributes of a commit such as hash, author, committer, author date, committer date, number of added lines, number of deleted lines, number of files changed, DMM unit size, DMM unit complexity, DMM unit interfacing, author email, and committer email.

Once all attributes are set, calling the build method returns a `ProjectCommit` object. This object is then categorized based on the "quality" of the included change and added to the `commit_analysis`.

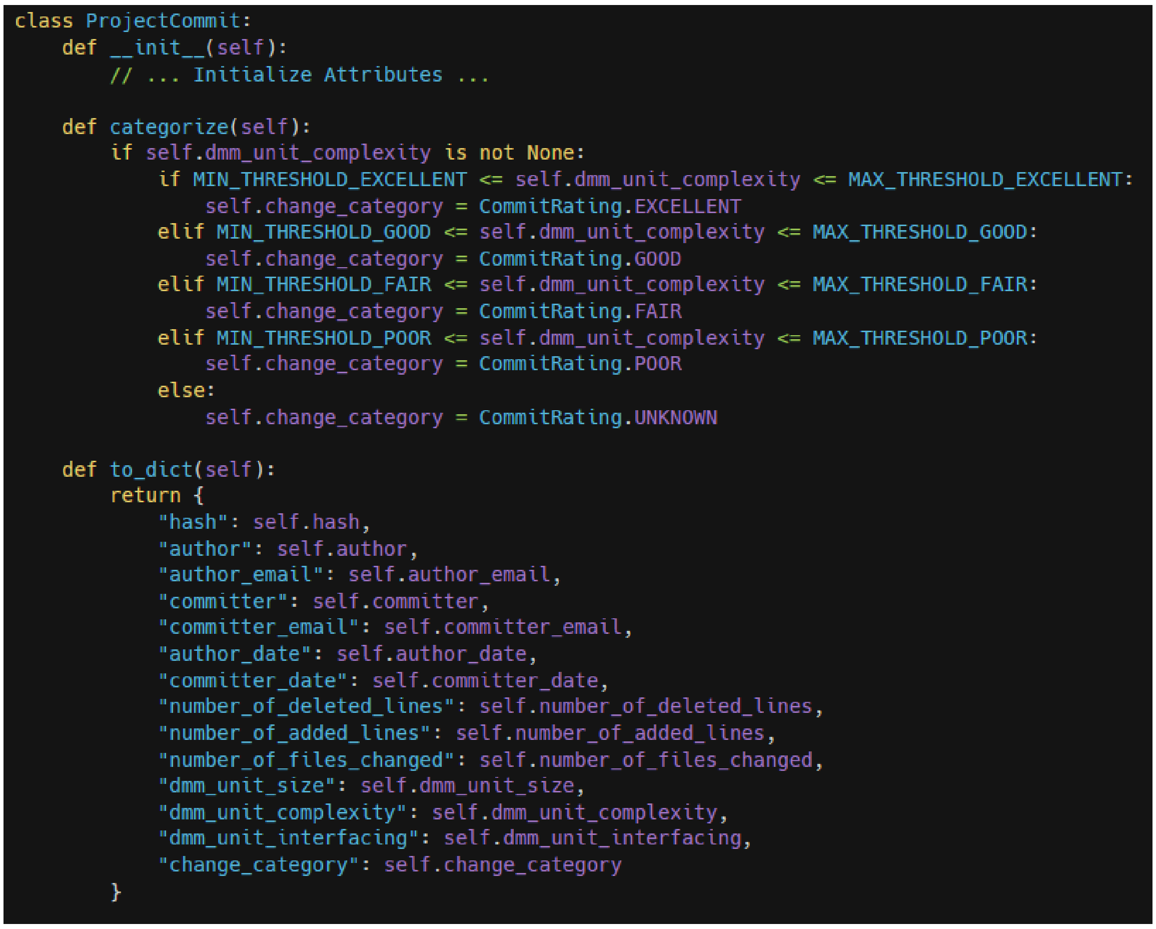


```
# A simple builder for ProjectCommit
class ProjectCommitBuilder:
    def __init__(self):
        self.commit = ProjectCommit()

    // ... Setters ...

    def build(self):
        return self.commit
```

Code Block 21 - Project Commit Builder Class (ProjectCommitBuilder.py)

4.4.4 The ProjectCommit Class

The `ProjectCommit` class in `ProjectCommit.py` represents a single commit in the repository. It contains various data such as hash, author, committer, author date, committer date, number of added lines, number of deleted lines, number of files changed, DMM unit size, DMM unit complexity, DMM unit interfacing, and change category.



```
class ProjectCommit:
    def __init__(self):
        // ... Initialize Attributes ...

    def categorize(self):
        if self.dmm_unit_complexity is not None:
            if MIN_THRESHOLD_EXCELLENT <= self.dmm_unit_complexity <= MAX_THRESHOLD_EXCELLENT:
                self.change_category = CommitRating.EXCELLENT
            elif MIN_THRESHOLD_GOOD <= self.dmm_unit_complexity <= MAX_THRESHOLD_GOOD:
                self.change_category = CommitRating.GOOD
            elif MIN_THRESHOLD_FAIR <= self.dmm_unit_complexity <= MAX_THRESHOLD_FAIR:
                self.change_category = CommitRating.FAIR
            elif MIN_THRESHOLD_POOR <= self.dmm_unit_complexity <= MAX_THRESHOLD_POOR:
                self.change_category = CommitRating.POOR
            else:
                self.change_category = CommitRating.UNKNOWN

    def to_dict(self):
        return {
            "hash": self.hash,
            "author": self.author,
            "author_email": self.author_email,
            "committer": self.committer,
            "committer_email": self.committer_email,
            "author_date": self.author_date,
            "committer_date": self.committer_date,
            "number_of_deleted_lines": self.number_of_deleted_lines,
            "number_of_added_lines": self.number_of_added_lines,
            "number_of_files_changed": self.number_of_files_changed,
            "dmm_unit_size": self.dmm_unit_size,
            "dmm_unit_complexity": self.dmm_unit_complexity,
            "dmm_unit_interfacing": self.dmm_unit_interfacing,
            "change_category": self.change_category
        }
```

Code Block 22 - Project Commit Builder Class (ProjectCommitBuilder.py)

The `categorize` method is responsible for categorizing the commit based on its DMM complexity. The method checks if the DMM unit complexity falls within certain thresholds. Depending on which range the DMM unit complexity falls into, the commit is categorized as ‘`EXCELLENT`’, ‘`GOOD`’, ‘`FAIR`’, ‘`POOR`’, or ‘`UNKNOWN`’.

The `to_dict` method converts the commit object to a dictionary for easier data manipulation and processing. This implementation allows for efficient categorization of commits based on their complexity. It provides valuable insights into the quality of changes made in each commit. This can be particularly useful for identifying problematic commits that may require further investigation or remediation.

5 Future Research

As the software development industry evolves, it is critical that CodeInspector stays flexible and adaptable to new challenges. In this section, we identify significant topics for future research and development that will expand CodeInspector's capabilities, making it more useful to software developers, engineers and researchers.

5.1 Data Enhancement

One of the key research areas is to broaden the scope of data analysis within the tool. This can be accomplished by adding new data sources and improving existing data analysis tools. Some potential approaches include:

- **Incorporate More Data Sources**
  Integrate data from various development platforms other than GitHub, such as GitLab, Bitbucket, and more. This will give a more complete picture of the software development process and quality.
- **Leverage External Services**
  Use external services such as Sonar and PyAssess to improve the analysis process. These services can offer specialized insights and analytics on code quality, performance, and security.

5.2 Granularity of Information

Future research can also focus on enhancing the granularity of data and insights given by CodeInspector to provide more specific and useful information to software developers and researchers. One way would be to extend the tool to provide comprehensive code snippets and changes for the analyzed project. This feature will allow developers to view the nature of code and commits helping them to further track the progress of code quality over time.

5.3 Improve Data Mining & Analysis

Advanced data mining and machine learning techniques can be combined to improve the precision and efficiency of the analytical process. We can improve the accuracy of hotspot discovery and quality assessment by implementing specific algorithms to identify complex patterns and relations inside repositories.

5.4 Integrate UoM Quality Dashboard & Validation

The seamless integration of CodeInspector with the Quality Dashboard at the University of Macedonia (UoM) can be a big step for future research:

- **Improve CodeInspector and QualityDashboard Compatibility:**
  Work on enhancing the compatibility and interaction between CodeInspector and UoM Quality Dashboard. This will streamline the workflow for software development teams and researchers by providing a unified platform for code quality assessment and project management.
- **Validation and Benchmarking:**
  The rich repository base of UoM Quality Dashboard will enable us to conduct extensive validation and benchmarking studies to evaluate the tool's effectiveness and impact in real-world scenarios. Case studies with software development teams and experts, as well as benchmarking against industry standards and best practices, can be used to examine the success of code quality assessment.

6 Conclusions

This thesis addresses the need for high code quality and hotspot priority in the context of GitHub repositories. As software engineering evolves, the development of more complex systems becomes the norm. As a result the importance of maintaining high-quality code is increasingly important. This centers around creating a tool that automates the process of prioritizing essential areas within codebases, hence contributing to the overall success of software projects. The proposed tool is capable of analyzing repositories written in various programming languages and extracts valuable information from repositories providing insights into the evolution and health of the codebase. A mechanism for identifying code files as "hotspots" and assigning them a priority was implemented to help development teams focus their efforts on the most critical areas of the codebase, where improvements are needed most urgently. The evaluation tool was created as a full-stack application that performs comprehensive software quality analysis. It combines hotspot prioritization, commit analysis, and data visualization to effectively display data. CodeInspector assists project managers and developers in making informed decisions by identifying hotspot files and distinguishing between good and bad commits.

In conclusion, this thesis not only promotes the discipline of software engineering by increasing code quality and hotspot prioritization methodologies, but also equips software engineers with a practical tool for assessing software quality. As the complexity and scope of software projects increase, the demand for effective and data-driven quality analysis tools becomes more important than ever. This study seeks to make a valuable contribution in addressing this need by ensuring that software stays adaptive, maintainable, and capable of efficiently serving its original function.

7 Bibliography

1. GitHub: Let's build from here. (n.d.). GitHub. Retrieved 1 November 2023, from https://github.com/
2. Git. (n.d.). Retrieved 1 November 2023, from https://git-scm.com/
3. Code search github. (2023, January 25). https://web.archive.org/web/20230125075800/https://github.com/search
4. ishepard/pydriller: Python Framework to analyse Git repositories. Retrieved 1 November 2023, from https://github.com/ishepard/pydriller
5. di Biase, M., Rastogi, A., Bruntink, M., & van Deursen, A. (2019). The delta maintainability model: Measuring maintainability of fine-grained code changes. 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), 113–122. https://doi.org/10.1109/TechDebt.2019.0003
6. O'Regan, G. (2018). Java programming language. In G. O'Regan (Ed.), The Innovation in Computing Companion: A Compendium of Select, Pivotal Inventions (pp. 171–174). Springer International Publishing. https://doi.org/10.1007/978-3-030-02619-6_35
7. Arnold, K., Gosling, J., & Holmes, D. (2006). The Java programming language (4th ed). Addison-Wesley.
8. Gosling, J. (2000). The java language specification. Addison-Wesley Professional.
9. Stack overflow developer survey 2023. (n.d.). Stack Overflow. Retrieved 1 November 2023, from https://survey.stackoverflow.co/2023/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2023
10. Most used languages among software developers globally 2023. (n.d.). Statista. Retrieved 1 November 2023, from https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/
11. Forbes. (n.d.). Forbes. Retrieved 1 November 2023, from https://www.forbes.com/
12. Belokrylov, A. (n.d.). *Council post: Why and how java continues to be one of the most popular enterprise coding languages*. Forbes. Retrieved 12 November 2023, from https://www.forbes.com/sites/forbestechcouncil/2022/04/06/why-and-how-java-continues-to-be-one-of-the-most-popular-enterprise-coding-languages/
13. *Pypl popularity of programming language index*. (n.d.). Retrieved 13 October 2023, from https://pypl.github.io/PYPL.html
14. *Tiobe index*. (n.d.). TIOBE. Retrieved 13 October 2023, from https://www.tiobe.com/tiobe-index/
15. Software tech news 6:2—Lessons learned in software quality assurance. (n.d.). Retrieved 13 October 2023, from https://www.eng.auburn.edu/~kchang/comp6710/readings/lessons.learned.in.SQA.htm
16. Gillies, A. (2011). *Software quality: Theory and management*.
17. Gomaa, H. (Ed.). (2011). Software quality attributes. In *Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures* (pp. 357–368). Cambridge University Press. https://doi.org/10.1017/CBO9780511779183.022
18. Digkas, G., Chatzigeorgiou, A., Ampatzoglou, A., & Avgeriou, P. (2022). Can clean new code reduce technical debt density? *IEEE Transactions on Software Engineering*, *48*(5), 1705–1721. https://doi.org/10.1109/TSE.2020.3032557

19. *ISO 25010*. (n.d.). Retrieved 12 November 2023, from https://iso25000.com/index.php/en/iso-25000-standards/iso-25010
20. *Quality attributes in software architecture | hackernoon*. (n.d.). Retrieved 12 November 2023, from https://hackernoon.com/quality-attributes-in-software-architecture-3844ea482732
21. *Code quality tool & secure analysis with sonarqube*. (n.d.). Retrieved 12 November 2023, from https://www.sonarsource.com/products/sonarqube/
22. s
23. Open Source UoM (n.d.). Retrieved 12 November 2023, from https://opensource.uom.gr/
24. *Software engineering group – software and data engineering lab*. (n.d.). Retrieved 12 November 2023, from https://sde.uom.gr/index.php/research-groups/software-engineering-group/
25. Apostolidis, G. D. (2023). Evaluation of Python code quality using multiple source code analyzers. http://dspace.lib.uom.gr/handle/2159/29041
26. Linter ide tool & real-time software for code | sonar. (n.d.). Retrieved 12 November 2023, from https://www.sonarsource.com/products/sonarlint/
27. Software engineering intelligence. (n.d.). Code Climate. Retrieved 12 November 2023, from https://codeclimate.com/
28. Pylint—Code analysis for Python | www.pylint.org. (n.d.). Retrieved 12 November 2023, from https://www.pylint.org/
29. *Continuous integration*. (n.d.). Martinfowler.Com. Retrieved 12 November 2023, from https://martinfowler.com/articles/continuousIntegration.html
30. The importance of pipeline quality gates and how to implement them. (n.d.). InfoQ. Retrieved 12 November 2023, from https://www.infoq.com/articles/pipeline-quality-gates/
31. di Biase, M., Rastogi, A., Bruntink, M., & van Deursen, A. (2019). The delta maintainability model: Measuring maintainability of fine-grained code changes. *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 113–122. https://doi.org/10.1109/TechDebt.2019.00030
32. Heitlager, I., Kuipers, T., & Visser, J. (2007). A practical model for measuring maintainability. *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, 30–39. https://doi.org/10.1109/QUATIC.2007.8
33. PyDriller Docs, Delta Maintainability (n.d.). Retrieved 12 November 2023, from https://pydriller.readthedocs.io/en/latest/deltamaintainability.html
34. Introducing the Eisenhower Matrix (n.d.). Retrieved 12 November 2023, from https://www.eisenhower.me/eisenhower-matrix/
35. Rossum, G. V. (2009, January 20). The history of python: A brief timeline of python. *The History of Python*. https://python-history.blogspot.com/2009/01/brief-timeline-of-python.html
36. Beginning python, advanced python, and python exercises. (2012, June 23). https://web.archive.org/web/20120623165941/http://cutter.rexx.com/~dkuhlman/python_book_01.html
37. *Download python*. (n.d.). Python.Org. Retrieved 12 November 2023, from https://www.python.org/downloads/
38. *Numpy*. (n.d.). Retrieved 12 November 2023, from https://numpy.org/
39. *Home—Vispy*. (n.d.). Retrieved 12 November 2023, from https://vispy.org/

40. *Pandas—Python data analysis library*. (n.d.). Retrieved 12 November 2023, from https://pandas.pydata.org/
41. *Fastapi*. (n.d.). Retrieved 12 November 2023, from https://fastapi.tiangolo.com/
42. *Features—Fastapi*. (n.d.). Retrieved 12 November 2023, from https://fastapi.tiangolo.com/features/
43. *Rest api documentation tool | swagger ui*. (n.d.). Retrieved 12 November 2023, from https://swagger.io/tools/swagger-ui/
44. React. (n.d.). Retrieved 12 November 2023, from https://react.dev/
45. *Docker: Accelerated container application development*. (2022, May 10). https://www.docker.com/
46. MySQL 8.0 Reference Manual. Oracle Corporation. Retrieved 12 November 2023 from https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html
47. Spadini, D. (n.d.). *Pydriller: Framework for Mining Software Repositories*. Retrieved 12 November 2023, from https://github.com/ishepard/pydriller