Received 24 September 2024, accepted 12 November 2024, date of publication 20 November 2024,
date of current version 29 November 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3503493

RESEARCH ARTICLE

Evaluating Python Static Code Analysis Tools Using FAIR Principles

HASSAN BAPEER HASSAN[1], **QUSAY IDREES SARHAN**[2], **AND ÁRPÁD BESZÉDES**[3]
[1]Medical Education Development Department, College of Medicine, University of Duhok, Duhok 42001, Iraq
[2]Computer Science Department, College of Science, University of Duhok, Duhok 42001, Iraq
[3]Software Engineering Department, Faculty of Science and Informatics, University of Szeged, 6720 Szeged, Hungary

Corresponding author: Hassan Bapeer Hassan (hassan.bapeer@uod.ac)

This work was supported in part by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund under Project TKP2021-NVA-09 and Project TKP2021-NVA; and in part by European Union within the Framework of the Artificial Intelligence National Laboratory under Project RRF-2.3.1-21-2022-00004.

ABSTRACT The quality of modern software relies heavily on the effective use of static code analysis tools. To improve their usefulness, these tools should be evaluated using a framework that prioritizes collaboration, user-friendliness, and long-term sustainability. In this paper, we suggest applying the FAIR principles—Findability, Accessibility, Interoperability, and Reusability—as a foundation for assessing static code analysis tools. We specifically focus on Python-based tools, analyzing their features and how well they align with FAIR guidelines. Our findings indicate that it is important to expand the FAIR principles to include thorough documentation, performance assessments, and robust testing frameworks for a more complete evaluation. As Internet of Things (IoT) applications and technologies become increasingly common, these tools must adapt to meet the unique challenges posed by complex and interconnected systems. Addressing these issues is vital for ensuring security and scalability within IoT environments. By implementing this FAIR-based approach, we aim to support the development of static code analysis tools that cater to the evolving needs of the software engineering community while ensuring they remain sustainable and reliable.

INDEX TERMS FAIR principles, IoT, python, software quality, static code analysis.

I. INTRODUCTION

The field of software engineering is continually evolving, focusing on enhancing the quality, reliability, and maintainability of software. Static code analysis tools are essential for detecting bugs and improving code quality. To maximize their effectiveness, these tools should be evaluated not only for technical capabilities but also for their alignment with principles like discoverability, accessibility, compatibility, and reusability. Python has become a key language in the Internet of Things (IoT) sector due to its straightforward syntax and rich library ecosystem, which supports tasks such as sensor interaction and cloud integration through tools like RPi.GPIO and paho-mqtt. Its versatility allows it to run on various platforms, from microcontrollers using MicroPython[1] and CircuitPython[2] to systems like Raspberry Pi. Additionally, Python's compatibility with popular hardware like Arduino and ESP32, along with libraries for data processing and machine learning such as NumPy and TensorFlow, positions it as a leader in IoT innovation, particularly in edge computing and AI applications. As reliance on Python for IoT increases, ensuring code quality and security becomes crucial, emphasizing the need for effective static code analysis tools.

This study explores the assessment of Python static code analysis tools through the perspective of FAIR principles (Findability, Accessibility, Interoperability, and Reusability) [1]. FAIR principles are initially designed for research data but also have shown growing significance in the realm of research software [2], [3], [4], [5]. By applying these principles to our assessment process of static code analysis

The associate editor coordinating the review of this manuscript and approving it for publication was Claudia Raibulet.

[1]https://micropython.org/

[2]https://circuitpython.org/

tools, we aim to encourage collaboration, cooperation, and the long-term viability of software tools, in the software engineering field. Checking how effective these tools are is extremely important for the IoT, where safety is key aspect. The IoT technology is becoming more common in our lives like smart grids, cars, and medical devices. These IoT devices are widespread and play essential roles, which makes them targets for attacks. In the case of IoT-enabled medical devices, static code analysis tools can spot vulnerabilities like improper input validation and buffer overflows, which could put patient safety at risk. Likewise, in smart grids, these tools play a vital role in protecting essential infrastructure by identifying potential security weaknesses in communication protocols and data processing algorithms. This emphasizes the need for good code analysis tools to identify and address such possible weaknesses [6], [7].

Measuring software quality using software metrics has been invaluable. With the increasing range of metrics extraction tools, it is vital to assess them based on principles. In this article, we have carefully selected a set of Python tools, considering their diverse features. Our goal is to provide an overview of each tool's advantages and drawbacks, empowering developers and corporations to select the most suitable tools for their particular needs.

The principal contributions of this paper are as follows:

- Providing a comparison of the available Python static analysis tools based on FAIR principles.
- Extending the FAIR principles with four additional indicators.
- Offering practical recommendations for selecting the most suitable tools for specific needs.
- Providing a discussion of the benefits of applying FAIR principles on the development of software tools.

The structure of this paper, designed to explore the application of FAIR principles to Python static code analysis tools, is as follows: Section II lays the groundwork by explaining the fundamentals of static code analysis, including its types, applications, and benefits. Section III focuses on the assessment of research software, proposing an adaptation of FAIR principles and introducing additional criteria specific to software tools. Section IV presents a detailed evaluation of selected Python static code analysis tools based on these adapted principles, explaining the selection process and scoring methodology. Section V provides a comparative analysis of the evaluated tools, highlighting their strengths and weaknesses. Section VI examines the top-scoring tools, discussing their advantages and best use cases. Section VII discusses the strengths and considerations of the top-scoring tools in greater depth. Section VIII offers practical recommendations for selecting the most suitable tools for specific needs. Finally, Section IX concludes the paper by reflecting on the importance of applying FAIR principles to software development, emphasizing how they enhance reproducibility, usability, and sustainability in the software engineering community.

II. STATIC CODE ANALYSIS

Static code analysis is used for examining software source code without execution to uncover potential defects, security weaknesses, and quality issues [8], [9]. It contrasts with dynamic analysis, which involves testing code during runtime.

A. KEY TYPES OF STATIC CODE ANALYSIS

Static code analysis encompasses several key types, each serving a distinct purpose in improving software quality and security. The following are the essential categories:

- Code Style Analysis
  It is the process of automatically examining source code to ensure it conforms to predetermined coding conventions, style guides, and formatting rules. This analysis improves code readability, consistency, and maintainability, making it particularly beneficial for large projects with multiple developers. Examples of rules enforced through code style analysis include consistent indentation, naming conventions for variables and functions, and line length limits [10].
- Code Quality Analysis
  Assessing code quality involves analyzing various characteristics that contribute to its overall health, maintainability, and robustness. The main goal is to identify potential problems early in the development cycle, which helps reduce future maintenance costs and prevents bugs from reaching production. Key metrics in this analysis include cyclomatic complexity, which measures the number of decision points in the code; code coupling, which assesses how interdependent different code modules are; code coverage, indicating the percentage of code executed by tests; and maintainability indices, which provide a score on how easy it will be to modify and maintain the code. Using these metrics is essential for ensuring that software remains functional and easy to work with over time [11], [12].
- Security Vulnerability Detection
  Static analysis techniques are essential for identifying common security flaws in code without requiring execution. By examining the code before it runs, developers can catch potential vulnerabilities early in the development process, which helps prevent costly security breaches and protects sensitive data. Some of the typical issues that static analysis can detect include SQL injection, cross-site scripting (XSS), buffer overflows, and improper input validation [13].
- Memory Leak Detection
  Memory leak detection is an important process aimed at identifying parts of the code where memory is allocated but not released afterward. If these leaks go unchecked, they can slowly degrade an application's performance or even cause crashes as the available memory runs out. By addressing memory leaks proactively, developers can ensure application stability, optimize memory usage,

and prevent resource depletion. Common causes of memory leaks include forgetting to deallocate objects after they are no longer needed, circular references that hinder garbage collection, and mistakes in custom memory management routines [14].

- Concurrency and Threading Analysis
  Concurrency analysis uses static analysis techniques to examine how multiple threads interact within an application. This type of analysis aims to identify potential synchronization issues that can be challenging to troubleshoot, such as race conditions and deadlocks. By detecting these problems early, developers can greatly improve the reliability of applications built for multithreaded environments. Common issues highlighted by concurrency analysis include insufficient locking mechanisms, unsafe data sharing between threads, and thread starvation [15].
- Compliance and Standards Checking
  Code compliance analysis involves a detailed examination of source code to ensure it meets established coding standards, regulations, and guidelines, which can be specific to an industry or unique to an organization. The main goal is to ensure code quality, minimize potential risks, and fulfill regulatory requirements, which is particularly crucial in software where errors can have serious consequences. Standards such as MISRA (commonly used in the automotive industry), HIPAA (related to healthcare data), and internal coding styles within large organizations all contribute to maintaining consistent and understandable code [16].
- Performance Optimization Analysis
  It utilizes static analysis techniques to pinpoint areas of code that could cause performance issues. The purpose of this analysis is to improve application responsiveness, reduce resource consumption, and enhance the overall user experience. This analysis can identify computationally expensive loops, inefficient algorithms, and potential areas of resource contention within the code [17].
- Dependency Analysis
  It examines the external libraries and components a project relies upon. The purpose of this analysis is to identify outdated dependencies and potential security vulnerabilities, ensuring software security, preventing compatibility issues, and keeping applications functioning smoothly with up-to-date dependencies [18].
- IoT Security
  The systematic evaluation of software source code employed in IoT systems without execution. The process helps to find potential security vulnerabilities, weaknesses, and quality issues within software components related to the IoT. Performed in a static mode, it allows to detect the security flaws that exist in the analyzed code before running it. As a result, such tools can significantly help decrease the risk of cyber-attacks and continuously maintain the overall security robustness of the entire IoT [6]. For example, in IoT systems like smart home devices, static code analysis tools can spot vulnerabilities such as weak cryptographic practices or hardcoded credentials, which helps to prevent unauthorized access.
- Documentation and Comments Analysis
  It involves assessing the presence, clarity, and usefulness of comments and inline documentation within source code. The aim of this analysis is to increase code understandability, reduce onboarding time for new developers, and facilitate effective maintenance over time [19].

B. APPLICATIONS OF STATIC CODE ANALYSIS

The applications of static code analysis cover all types of software/hardware solutions such as IoT, AI, etc.

- Early Defect Detection
  Static analysis can uncover bugs, security vulnerabilities, and instances of poorly written code ("code smells") during early development, reducing the cost of remediation and mitigating risks. This capability is supported by research into various static analysis tools.
- Code Quality Improvement
  Static analysis fosters code readability, maintainability, and adherence to coding standards, enhancing long-term project health.
- Security Vulnerability Detection
  Static analysis can be essential for finding and addressing potential security holes before deployment, protecting applications from exploits. The importance of static analysis in security is discussed in multiple studies [20].
- Compliance
  Static analysis supports adherence to industry-specific regulations (e.g., PCI DSS and HIPAA) and security guidelines.

C. ADVANTAGES OF STATIC CODE ANALYSIS

Static Code Analysis is instrumental in modern software development for several key reasons [8]:

- Cost-Effectiveness
  Static analysis offers a more efficient and scalable approach to finding common defects compared to manual inspection. This is supported by comparative studies on different static analysis tools.
- Shift-Left Approach
  It aligns with the "shift-left" paradigm, emphasizing the importance of integrating testing and quality assurance early in the development process. Various research papers highlight this aspect of static analysis.
- Automation
  Static analysis automates repetitive tasks within analysis, freeing up developers' time, and ensuring greater consistency.

- Objectivity
  Static analysis provides an unbiased perspective on code quality, mitigating some of the subjectivity inherent in manual code analysis and evaluation.

D. KEY CHALLENGES IN STATIC CODE ANALYSIS

- False Positives and Negatives
  A key challenge lies in balancing sensitivity to reduce false alarms (where the tool indicates non-existent issues) without missing critical issues (false negatives). Achieving this balance is essential for building trust in the tool's output [9], [21].
- Scalability
  Ensuring static analysis tools can efficiently analyze large, complex codebases without significant performance impacts is crucial, especially in modern development environments with vast amounts of code [22].
- Adapting to Modern Languages and Frameworks
  Static analysis tools need to constantly adjust to changes in programming languages, frameworks, and architectures. Keeping up with the latest developments helps these tools analyze modern coding methods and find security flaws that come with new technologies.
- Complexity of Analysis
  Even with static analysis tools, it can be challenging to find all potential software errors or weaknesses in complex coding, especially in large projects. This is because the complexity of the code can limit the tool's ability to thoroughly analyze all possible paths and identify hidden issues, especially when the code logic is complex or uses uncommon coding patterns.
- Limited Contextual Understanding
  Static analysis tools frequently work without a thorough understanding of the application's unique business logic or intended behavior. This can make accurate analysis and fault assessment more challenging, possibly leading to false positives or misinterpretation of issues [23].
- Interprocedural Analysis
  Static analysis tools face challenges when examining code that is spread across multiple functions or modules. It is essential to understand how data and code flow connect these code sections to identify specific types of vulnerabilities [24].

III. EVALUATION GUIDELINES FOR RESEARCH SOFTWARE

Assessing research software is challenging due to its constant evolution and the complex interplay between its source code, capabilities, and information. Although some initial frameworks exist, the field lacks comprehensive and globally recognized standards to guarantee that software is: reproducible, usable, and sustainable.

A. ADAPTING THE FAIR PRINCIPLES

Originally designed for scientific data, the FAIR principles have gained wide recognition. However, they need to be adjusted to be practical for use with research software. This need for modification is due to software's unique properties like executability, versioning, and functionality, as highlighted by [2] and [5]. The importance of adapting FAIR principles specifically for life sciences research software is further underscored to promote trustability and reproducibility [2]. Both works emphasize the importance of ongoing community discourse to define precise FAIR metrics tailored to research software. To guide our evaluation, we summarized the core FAIR principles and our additions in Table 1.

B. METADATA, TOOL DESIGN, AND QUANTITATIVE ASSESSMENT

Balancing metadata-related quality standards with expectations for software functionality is a key concern. This tension, and the need for guidelines to promote future usability, is investigated by Soha [4]. Furthermore, the potential for streamlining large-scale, automated assessment of software quality using FAIR indicators is explored by del Pico et al. [5]. However, limitations exist due to a lack of standard identifiers and inconsistent metadata across repositories.

C. NEW PRINCIPLES

We extended existing principles outlined in [3], [4], and [5] with four additional indicators, listed below. Our additions to the reusability guidelines emphasize the importance of documentation, performance clarity, and testing for software usability and reproducibility.

- R1.6: Open Issues and Accessibility: To promote transparency and facilitate troubleshooting, we advocate for thorough issue documentation. This principle emphasizes the value of easily accessible issue tracking within the project's code repository or website. It can be interpreted as: "Open issues are thoroughly documented and easily accessible within the project repository or website."
- R5: Performance Clarity: We believe users should have readily available performance metrics to make informed decisions about software adoption. This principle emphasizes the importance of having clear performance measurements for software, specifically its runtime, memory consumption, and computational complexity. Users require this information to check if the software works with their hardware and to ensure that the results they get can be reproduced. To maintain accuracy in these measurements, it is important to update them regularly and compare them against actual usage patterns through benchmarking.
- R6: Comprehensive Testing: Thorough test suites are essential to ensure that software is correct, reliable, and reproducible. This means that testing should address not only the typical scenarios users might encounter, but also those unusual edge cases that could cause problems. It can be stated as: "The software includes

TABLE 1. FAIR principles and descriptions.

Principle	Description
F1	Each released version of software, along with its related metadata, is assigned a distinctive, global, and long-lasting identifier.
F1.1	The software has a unique name to identify it.
F1.2	A scheme is used to uniquely and properly identify the software version.
F2	Comprehensive and detailed metadata is used to describe the software.
F3	The metadata explicitly includes distinctive identifiers for all versions of the described software, ensuring clarity and precision.
F4	The software, along with its related metadata, is incorporated into a searchable registry for software.
A1	The software, along with its related metadata, can be accessed through its identifier using a standardized communication protocol.
A1.1	The protocol is open, free, and can be implemented universally.
A1.2	The protocol enables authentication and authorization procedures as needed.
A1.3	A set of instructions and other necessary information the user can follow to build the software is available.
A1.4	Test data is available.
A1.5	Source code of the software is available.
A2	Metadata for the software remains accessible even if the software itself is no longer available.
A2.1	Metadata of previous versions is available.
A2.2	Previous versions are available.
A3	No restrictions exist to access the software.
A3.1	The software can be used without registration.
A3.2	The software can be used in a free operating system.
A3.3	Versions of the software for several operative systems are available.
I1	The software and its associated metadata utilize a formal, accessible, shared, and widely applicable language to enhance machine readability and facilitate data exchange.
I1.1	Input and output data types are formally specified and related to accepted ontologies.
I1.2	APIs (Rest, libraries) are documented in a standard framework (OpenAPI, WES...).
I1.3	Input/output data are specified using verifiable schemas (e.g. XDS, Json schema, ...).
I1.4	The software allows users to choose among various input/output data formats, or provide the necessary tools to convert other common formats into the supported ones.
I1.5	The software provides provenance information according to accepted standards (PROV).
I2	The software can be deployed in a format to be included in pipelines.
I2.1	The software has API/library versions to be included in users' pipelines.
I2.2	The software can be deployed in e-infrastructures (e.g. Galaxy).
I3	A proper documentation on the software's dependencies as well as mechanisms to obtain them is available.
I3.1	The software includes details about dependencies.
I3.2	The software includes its dependencies or mechanisms to access them.
I3.3	The software is distributed via a dependencies aware system.
R1	The software, along with its associated metadata, is extensively described with a variety of precise and pertinent attributes.
R1.1	The software, along with its associated metadata, is endowed with individual, transparent, and easily accessible usage licenses that align with the software dependencies.
R1.2	The software metadata provides in-depth information about its origin, with the level of detail determined by consensus within the community.
R1.3	The metadata and documentation for the software adhere to community standards relevant to the domain.
R1.4	Documentation and associated metadata for the software contain the current contact details of the authors, enabling them to provide support for their software.
R1.5	Examples of use cases are provided.
R1.6	Open issues are thoroughly documented and easily accessible within the project repository or website.
R2	The software is regularly maintained and is compatible with the latest versions of the programming language in which it was developed.
R3	A contributors policy exists.
R3.1	A document stating the contributors policy exists.
R3.2	Credit for contributions is provided.
R4	Provenance is available.
R4.1	The software follows a version-control system.
R4.2	The software follows a defined and documented release policy.
R4.3	Metadata of previous versions is available.
R5	Clear performance metrics (e.g., runtime, memory usage, computational complexity) are provided to enable users to assess the software's suitability for their hardware and ensure the reproducibility of results.
R6	The software includes comprehensive test suites, covering a wide range of use cases and edge cases, to ensure its correctness, reliability, and reproducibility.
R6.1	Documentation on how to run tests is included, aiding users who may want to replicate results on their systems.

comprehensive test suites, covering a wide range of use cases and edge cases, to ensure its correctness, reliability, and reproducibility.''

- R6.1: Test Execution Guidance: To ensure that results can be reproduced, it is important to offer clear and detailed guidelines for executing tests. This principle highlights the need to provide instructions on how to run the available test suites. It can be interpreted as: ''Include documentation on how to run tests, aiding users who may want to replicate results on their systems.'' This documentation should also outline the process for contributing new test cases or updating existing ones.

IV. FAIR-BASED EVALUATION

To evaluate static code analysis tools against the FAIR principles, we curated a comprehensive collection of Python tools. Our tool selection criteria were based on studying academic literature, online repositories, and relevant resources. The process began by exploring popular open-source projects, software repositories, and GitHub to discover both well-known and emerging tools, leveraging their repositories and community discussions. The output is a preliminary list of tools that then has been expanded by conducting a thorough literature analysis, where relevant research articles provided additional tool suggestions.

To capture a broader range of tools, a regular Google search was performed, which was particularly useful in identifying corporate and private tools not typically covered in scholarly articles. We included both popular and lesser-known tools from different time periods to provide a comprehensive evaluation. The final selection of tools for this study is listed below with the relevance of each tool to the requirements presented in Table 1:

- Prospector [25]: is a tool for Python static analysis that finds errors, potential code quality issues, standard violations, and complexity issues (F1). It makes regular releases with clear versioning (F2, F3). It is straightforward to install with standard Python package management (A1) and will generally integrate into most code editors or build pipelines. While it may not have a standalone metadata registry (A2), configuring its various options will also generally allow developers to customize it to various use-cases (I1). Released under a GNU General Public License v2.0 (R1), its GitHub repository is very active showing good transparency of its development history and contributions (R1). It will generally rely on other Python libraries for its various types of analysis (R2).
- Pylint [26]: is a respected Python linter that specializes in finding errors, enforcing style rules (similar to PEP 8 guidelines), and detecting code issues (''code smells''). It provides detailed code quality metrics and a rating system for tracking improvements. Pylint is updated regularly with clear versioning (F2, F3). It is easy to install through Python's package management system (A1) and integrates well with code editors, build systems, and offers a command-line interface. While it lacks a standalone metadata registry (A2), Pylint's extensive customization options and limited API-based extensibility make it compatible with various other tools (I1).It is licensed under GPL-2.0, making its development history and contributions fully accessible through its active GitHub repository. It depends on the astroid [26] package as a core component (R2).
- Pygount [27]: is a Python code analysis tool specializing in basic source lines of code (SLOC) metrics, alongside code comment volume and language distribution analysis (F1). It leverages the Pygments [28] library for language detection, supporting a wider variety of file types than some similar tools (F1). pygount benefits from regular releases with clear versioning (F2, F3). Easily installed via standard Python package management (A1), it offers both a command-line interface and the ability to analyze remote Git repositories. While a standalone metadata registry might be absent (A2), pygount has a limited API for potential integration into custom workflows (I1). Released under the BSD License (R1), its development history is accessible on platforms like GitHub (R1). pygount relies on the Pygments package as a core dependency (R2).
- SonarQube's [29] Python Analyzer (Sonar Python): is a component of the larger SonarQube software quality platform. It focuses on static analysis for Python projects, promoting code quality and security (F1). This component has a well-documented release history through the main SonarQube releases (F2, F3). SonarQube is easily installed for on-premises use or offers cloud-based options (SonarCloud) (A1). While internal project metadata might be stored, it likely does not provide a public metadata registry (A2). SonarQube and its components are released under the GNU LGPLv3 (R1) and its development history is accessible through the main SonarSource repositories (R1). The Python analyzer integrates with SonarLint for in-IDE support and leverages typeshed for improved analysis.
- Radon [30]: is a Python static analysis tool specializing in code complexity metrics. It calculates Cyclomatic Complexity, Halstead metrics, and a custom Maintainability Index to assess code understandability and potential refactoring needs (F1). Radon benefits from regular releases with clear versioning (F2, F3). Easily installed via standard Python package management (A1), it offers both a command-line interface and an API for integration into workflows. While it might not have a standalone metadata registry (A2), Radon's focus on well-defined metrics supports interoperability with other analysis tools (I1). Released under the MIT license (R1), its development history is accessible on platforms like GitHub (R1). Radon relies on the mando [30] and colorama [31] packages as dependencies (R2).

- McCabe [32]: is a highly specialized Python static analysis tool designed to calculate McCabe's Cyclomatic Complexity metric, which offers insights into code understandability and maintainability (F1). Primarily used as a Flake8 plugin, McCabe flags functions exceeding a user-defined complexity threshold (F1). It has well-defined releases with versioning (F2, F3). Easily installed via standard Python package management (A1), its primary user interface is through integration with Flake8. While a standalone metadata registry might be absent (A2), its focus on a single, quantifiable metric enhances potential uses in conjunction with other analysis tools (I1). Released under the MIT license (R1), its development history is transparent on platforms like GitHub (R1).
- PyLint-MCCabe [33]: is a legacy plugin for the Pylint linter, designed to integrate McCabe Cyclomatic Complexity calculations into Pylint's analysis output (F1). While its documentation is limited and recent releases are absent (A1, F2), it might serve as a historical reference point for assessing complexity measurement techniques (F3). The plugin is easily installed alongside Pylint and McCabe dependencies via standard Python package management (A1). Lacking a standalone metadata registry (A2), its primary user interface is through Pylint. Released under the MIT license (R1), its development history might be available on platforms like GitHub but likely shows limited recent activity (R1).
- Decomplexator [34]: a Python code analyzer, combines two metrics: Cyclomatic Complexity (for code structure) and Cognitive Complexity (for readability) (F1). This holistic approach aims to improve code comprehension assessments. The tool offers easy installation via Python package management and a command-line interface. It lacks full documentation (A1) and recent updates (F2,F3). However, it can track analysis results for comparison. Decomplexator is open-source under the MIT license, and its development details are accessible on GitHub platform (R1).
- Cognitive-complexity [35]: is a Python static analysis tool designed to calculate a Cognitive Complexity metric for functions (F1). This metric aims to assess code readability and maintainability by taking into account control flow, nesting, and recursion complexity factors. It offers limited documentation primarily within its Readme (A1) and has regular releases with clear versioning (F2, F3). Easily installed via standard Python package management (A1), it provides both a command-line interface and an API for programmatic use. While it might not have a standalone metadata registry (A2), it has integrations like the flake8-cognitive-complexity extension, enhancing its interoperability (I1). Released under the MIT license (R1), its development history is accessible on platforms like GitHub (R1).
- Flake8-cognitive-complexity [35]:
is an extension for the Flake8 linter that measures the cognitive complexity of Python functions in an effort to measure the readability and maintainability of code (F1). It has a history of releases, though recent activity might be lower (F2, F3). It runs through a command-line interface and is easily installed using standard Python package management (A1). It is also integrated directly within Flake8 workflows. As a Flake8 extension, it does not have a standalone metadata registry (A2). Its development history may be accessible on websites such as GitHub, but it may only display a small number of recent contributions. It is licensed under the MIT license (R1). Naturally, it depends on Flake8 itself to function (R2).
- Pyan [36]: is a Python library that focuses on conducting static code analysis to create call dependency graphs. These graphs are instrumental in mapping out the interconnections between various functions and methods, thus enhancing the understanding of the code structure (F1). Despite its utility, the module's documentation could be more comprehensive (A1). It boasts a series of releases over time, although there might be a noticeable dip in the frequency of recent updates (F2, F3). The installation process for Pyan is straightforward, utilizing common Python package management systems (A1), and it includes a command-line interface for ease of use. However, it appears to lack an independent metadata registry (A2). Pyan is distributed under the GPL-2.0 license (R1), and while its development history can likely be traced on platforms such as GitHub, recent contributions seem to be sparse (R1). For generating visualizations, Pyan relies on GraphViz (R2).
- Lizard [37]: is a flexible static code analysis tool that focuses on calculating code metrics and finding code duplication (F1). It supports multiple programming languages and can calculate various metrics, including NLOC, Cyclomatic Complexity, token count, and parameter count (F1). Lizard has an extensive release history (F2, F3) and is easy to install using standard Python package management (A1). While it lacks a standalone metadata registry (A2), it features a command-line interface and offers some extensibility through an API. Released under a Freeware license (R1), its development history can be tracked on community repositories like GitHub (R1).
- Wily [38]: is a command-line static analysis tool that focuses on tracking and reporting a host of different code complexity metrics for Python projects, including Lines of Code, Cyclomatic Complexity, and Maintainability Index (F1). Wily uses Git to store historical analysis, which allows trends to be visualized across revisions (F1). It has a well-documented release history (F2, F3); it can be installed as standard using either Python package management or Conda (A1). While it does have

a command-line interface and does not have a standalone metadata registry (A2), Wily is licensed under Apache-2.0 (R1). Its development history is available through its repository on GitHub (R1).

- Pyanalyzer [39]: is a legacy Python static analysis tool focusing on code metrics calculation. It provides metrics like Lines of Code, function and class complexity, and structural metrics related to nesting and coupling (F1). While its documentation appears somewhat limited (A1), it has a release history, though recent activity might be lower (F2, F3). Easily installed from its source code (A1), Pyanalyzer operates via a command-line interface. As a legacy tool, it likely lacks a standalone metadata registry (A2). Pyanalyzer is released under the Apache Software License (R1), and its development history might be available on platforms like GitHub, but updates might be infrequent (R1).
- Codemetrics [40]: is a Python library designed to help developers gain insights into code metrics and code evolution by utilizing source control history, such as Git or Subversion, along with tools like Pandas, Lizard, and cloc (F1). It provides valuable metrics like Lines of Code, Cyclomatic Complexity, code age, and the number of lines changed per revision (F1). The library has a clear release history (F2, F3) and is straightforward to install using Python's standard package management tools (A1). While it does not include a standalone metadata registry (A2), it offers a Python library API that can be customized and integrated into projects. Codemetrics is licensed under MIT, and its development history is openly accessible on GitHub (R1). To make full use of its capabilities, you need to install dependencies like Pandas, Lizard, cloc, and a source control management tool such as Git or Subversion (R2).
- Pynocle [41]: is a Python library focused on calculating code metrics. It supports metrics like Cyclomatic Complexity, Lines of Code, test coverage, dependency graphs, and coupling (F1). While its documentation is somewhat limited (A1), Pynocle has a release history and can be installed relatively easily from its source code (A1). The library includes a Python API, allowing integration with other code analysis workflows. Being a legacy tool, it likely does not include a standalone metadata registry (A2). Pynocle is distributed under the MIT license (R1), and its development history may be available on platforms like Google Code, though updates are probably infrequent (R1). Its functionality depends on external tools such as GraphViz, Docutils, Coverage, and NumPy (R2).
- Multimetric [42]: is a Python-based tool designed for ambitious static analysis across a wide range of programming languages. It calculates a diverse set of code metrics, including comment ratios, Cyclomatic Complexity, Halstead metrics, maintainability, and even integrates with Pylint (F1). Multimetric has well-documented releases (F2, F3) and is easily installed using standard Python package management (A1). While it might not have a standalone metadata registry (A2), it offers a command-line interface. Multimetric is released under the zlib/libpng license (R1) and its development history is accessible on platforms like GitHub (R1). It primarily depends on Python, Chardet, and Pygments (R2).
- Metrics-felbeaver [35]: is a Python package focused on static code analysis for complexity assessment. It calculates a range of metrics, including Lines of Code (with variations), Halstead metrics, McCabe's Cyclomatic Complexity, and Henry-Kafura Information Flow Complexity to analyze module dependencies (F1). The project provides documentation (A1), has a history of releases (F2, F3), and is easily installed using standard Python package management (A1). While it might not have a standalone metadata registry (A2), it offers a command-line interface and a Python API for integration or extension. metrics-falbeaver is released under the MIT license (R1) and its development history might be available on platforms like GitHub, though recent activity could be lower (R1).

This deliberate curation of tools empowers us to evaluate a wide spectrum of capabilities essential for a comprehensive FAIR assessment of static code analysis tools for Python.

V. COMPARISON

Analysis reveals that SonarQube (score: 39), Pygount (score: 34.5) and PyLint (score: 33), as front runners within this dataset as shown in Table 2. To arrive at these scores, each tool was assessed against FAIR principles using the following scoring scheme:

- Yes = 1
- Y/P (Yes/Partially) = 0.5
- No = 0
- N/A (not applicable) = Excluded

TABLE 2. Tool evaluation scores.

Tool	Overall Fair Score
SonarQube	39
Pygount	34.5
PyLint	33
Prospector	31.5
Radon	31.5
Codemetrics	31.5
Py0cle	31.5
Cognitive-complexity	30.5
Multimetric	30
Flake8-CC	29.5
Wily	29.5
McCabe	28
PyLint-MCCabe	28
Lizard	28
Metrics	28
Pyan	26
Pyanalyzer	26
Decomplexator	25

This assessment highlights general adherence to Findability (F), Accessibility (A), Interoperability (I), and Reusability (R) principles, suggesting a focus on usability, discoverability, and long-term value. The limited focus on 'identification' and 'archiving' (evident in the Decomplexator score of 25) undermines long-term reproducibility within Python analysis workflows. This could potentially hinder the reliability of findings over time.

This tool prioritizes being easy to find, use, and integrate into existing Python workflows, promoting knowledge sharing throughout development communities. However, its limited emphasis on archiving (A) and identification (F) creates challenges for ensuring long-term reproducibility. This poses particular risks within scientific Python environments, where rigorous tracking of methods and data provenance is foundational. Without robust archiving and identification practices, the ability to replicate results and track specific software versions used in past analyses diminishes. For example, a researcher might struggle to exactly recreate an analysis conducted several years prior.

Evaluating these tools in depth suggests that reproducible research software relies on the following key principles:

* **Thorough Documentation (R1.6):** The strong emphasis on documenting open issues within these tool repositories aligns with best practices for open-source development. This supports transparency and facilitates deeper understanding of the analysis capabilities within software workflows.

* **Performance Transparency (R5):** The lack of performance metrics across most tools hinders informed tool selection and compromises the reproducibility of results, particularly when analyzing large-scale datasets and software. Without these metrics, users risk wasting time and resources on unsuitable tools

* **Test Suites and Guidance (R6, R6.1):** While most tools include test suites, many lack clear documentation on their execution. This limits users' ability to independently verify results and contribute to development, hindering both reproducibility and trust within software communities.

VI. STRENGTHS AND CONSIDERATIONS FOR THE TOP-SCORING TOOLS

* **SonarQube:** This tool prioritizes in-depth analysis and comprehensive reporting for scrutinizing Python code. However, this focus comes at the cost of limited performance metric reporting (R5). This trade-off should be carefully evaluated based on individual project requirements.

* **PyLint:** PyLint's focus on style aligns with the Python community's emphasis on readable, maintainable code. However, for projects with complex use cases, careful evaluation of its test suite and documentation (R6, R6.1) is warranted.

* **Pygount:** While Pygount offers a narrower feature set than SonarQube, its strong score highlights its capabilities. Its transparent approach to open issues (R1.6) could make it a good fit for projects where community engagement or customization is a priority. This evaluation highlights the need for Python static code analysis tools to strike a better balance between delivering immediate benefits—such as findability, accessibility, interoperability, and reusability—while also emphasizing the importance of identification (F) and archiving (A) principles to ensure long-term value. Clear performance metrics (R5), reliable test suites, and accessible testing instructions (R6.1) are essential for maintaining reproducibility and supporting detailed scientific workflows over time. The diverse range of tool performance, from the limitations of Decomplexator to the strengths of the top performers, underscores the importance of thoughtful tool selection. To make the best choice, it is critical to rigorously evaluate the specific needs of each Python project, including the quality of documentation, resource demands, and testing capabilities.

VII. DISCUSSION

In this section, we examined the selected Python code analysis tools in relation to the extended FAIR principles, summarizing our findings in Table 3. A key focus was on the findability of these tools. For F1, most tools met the requirement of having a unique name but lacked distinctive identifiers for both the software and its metadata, leading to a "Yes/Partially" designation. Unfortunately, F2 and F3 were not met by most tools, as many are hosted on platforms like GitHub that provide basic metadata but do not offer the detailed identifiers necessary for tracking all software versions. Nevertheless, these tools generally scored well for F4 due to their searchability. Additionally, it's crucial for research software to have unique and persistent identifiers (PIDs) for each version to ensure reproducibility and track changes over time. Platforms like Zenodo address this need by assigning Digital Object Identifiers (DOIs) to software releases, guaranteeing long-term access and linking all versions of a software release, which helps researchers reliably cite and access the tools they need over time [2]

Accessibility ratings were positive. Most tools leverage standard HTTPS protocols (A1, A1.1) and offer unrestricted access (A3). The availability of metadata for previous versions (A2, A2.1) could not be fully assessed since most tools remain actively maintained.

Interoperability proved challenging. While most tools utilize metadata for essential functionality, it rarely adheres to formal, shared languages (I1). Similarly, the lack of controlled vocabulary support for source code or metadata led to "N/A" ratings for I2-related principles.

Reusability guidelines were partially met. Licensing information (R1.1) and documentation (R1.3) were generally present. However, metadata detailing the software's origins (R1.2) was often insufficient. We also assessed additional reusability indicators. Tools varied in how thoroughly they addressed open issue tracking (R1.6). Additionally, while most demonstrated regular maintenance (R2), the explicit provision of performance metrics (R5) and comprehensive test suites (R6) were less common.

Importantly, our analysis highlights the importance of considering a tool's overall FAIR compliance when making

TABLE 3. Applying FAIR principles to python code analysis tools.

Principle	Prospector	PyLint	Pygount	SonarQube	Radon	McCabe	PyLint-MCCabe	Decomplexator	Cognitive-complexity	Flake8-CC	Pyan	Lizard	Wily	Pyanalyzer	Codemetrics	Pynocle	Multimetric	Metrics
F1	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P
F1.1	No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	Yes	No	No
F1.2	Yes	Yes	Yes	Y/P	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
F2	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
F3	No	No	No	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No
F4	No	No	No	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No
A1	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
A1.1	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
A1.2	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
A1.3	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
A1.4	No	No	No	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No
A1.5	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
A2	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
A2.1	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
A2.2	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
A3	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
A3.1	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
A3.2	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
A3.3	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
I1	N/A	N/A	N/A	Yes	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
I1.1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
I1.2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
I1.3	N/A	N/A	Yes	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
I1.4	N/A	N/A	Yes	Y/P	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
I1.5	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
I2	Y/P	Y/P	Yes	Yes	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	No	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P
I2.1	No	Y/P	Yes	Yes	Yes	No	No	No	Yes	No	No	Yes	Yes	No	Yes	Yes	No	Yes
I2.2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
I3	No	No	No	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	Y/P
I3.1	Yes	Yes	No	Yes	Yes	No	Y/P	No	No	No	Y/P	Y/P	No	Yes	Yes	Yes	Yes	No
I3.2	Yes	Yes	No	Yes	Yes	No	No	No	No	No	Y/P	No	No	Yes	Yes	Y/P	Yes	No
I3.3	Yes	Yes	Yes	No	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	No	Yes	Yes	Yes	Yes
R1	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
R1.1	Y/P	Y/P	Y/P	Yes	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P	Y/P
R1.2	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
R1.3	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
R1.4	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
R1.5	N/A	N/A	Yes	Yes	Yes	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
R1.6	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
R2	Yes	Yes	Yes	Yes	Yes	Yes	No	Y/P	Yes	Yes	Yes	Yes	Yes	Y/P	Yes	Yes	Y/P	No
R3	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Yes
R3.1	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Yes
R3.2	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
R4	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
R4.1	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
R4.2	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
R4.3	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
R5	No	No	No	Y/P	No	No	No	No	No	No	No	No	No	No	No	No	No	No
R6	Yes	Yes	Yes	Yes	Yes	Y/P	No	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Y/P	Yes	No
R6.1	Yes	Yes	Y/P	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No

selections for your projects. Additionally, we observed that tools like Prospector and Pylint generally satisfy most FAIR principles. These tools stand out with features like clear versioning, regular maintenance, and readily available documentation. At the same time, legacy tools like PyLint-MCCabe or Pynocle often lacked recent updates or comprehensive metadata. This indicates an opportunity to either update these tools to meet current standards or carefully consider alternatives for new projects.

The dynamic nature of software engineering underscores the necessity for adaptable evaluation frameworks. Our study exemplifies the benefits of integrating FAIR principles into this evolving field. Future research should consider further methodological refinements and expansions, such as incorporating user-centric metrics or domain-specific considerations, which could offer tailored insights for various areas of software analysis.

Static code analysis tools must evolve to meet the diverse needs of IoT devices, which vary significantly in their capabilities and functions. Many IoT-enabled systems currently rely on these tools, but they must continuously adapt to tackle emerging threats and changing regulatory standards. For example, software in medical devices such as insulin pumps and heart monitors can become vulnerable as cyber-attack methods evolve or compliance requirements shift. Similarly, in industrial IoT (IIoT) environments, static analysis tools are used to identify weaknesses in automated control systems. However, as these systems become more complex and interconnected, there is a pressing need for advanced tools that can detect new vulnerabilities and ensure robust security measures. These tools should not only adhere to FAIR principles but also be adaptable enough to fit various operational contexts while maintaining high security standards. This flexibility is essential for preserving the integrity and reliability of IoT ecosystems. Improving these tools to address the unique demands of IoT—while remaining aligned with FAIR principles—is critical for building secure, efficient, and sustainable infrastructures.

A useful approach for the future would be to design a web application that helps users select the most suitable static code analysis tools for their needs. Users could start by choosing their programming language and specifying the type of analysis they need, such as code slicing, error detection, finding code smells, or detecting memory leaks. Based on these inputs, the application would recommend tools and evaluate them using FAIR principles, providing clear and transparent scores. This intuitive platform would simplify decision-making for developers and researchers, making it easier to identify the best tools for their projects quickly and effectively.

VIII. TOOL SELECTION FOR SPECIFIC USE CASES

Choosing the right static code analysis tool requires a clear understanding of your project's specific needs and goals. This section outlines several common use cases along with recommended tools based on their features and FAIR evaluation scores shown in Table 2. Tools like SonarQube and PyLint, which have high FAIR scores and robust functionality, are great options for various scenarios. For more specialized tasks, consider tools like Radon for performance optimization and Pygount for documentation analysis. Although FAIR scores are helpful, it is important to take into account the unique context of your project to make the best choice.

A. USE CASE 1: CODE QUALITY IMPROVEMENT

1) REQUIREMENTS

- Improve code readability and maintainability.
- Enforce coding standards and conventions.
- Detect potential code smells and identify opportunities for refactoring.

2) RECOMMENDED TOOLS

- **PyLint**: With a high FAIR score, PyLint is effective in delivering detailed metrics on code quality and enforcing style guidelines, making it particularly suitable for projects that prioritize high standards of readability and maintainability.
- **Prospector**: This tool combines various analysis components to deliver a thorough evaluation of code quality, including style and complexity metrics, which contribute to substantial improvements in overall code quality.

B. USE CASE 2: SECURITY VULNERABILITY DETECTION

1) REQUIREMENTS

- Identify potential security vulnerabilities in the code.
- Ensure adherence to security best practices.
- Detect common security flaws, including SQL injection, cross-site scripting (XSS), and buffer overflows.

2) RECOMMENDED TOOLS

- **SonarQube**: SonarQube is known for its strong security analysis features and compliance checks, making it a prominent tool for finding vulnerabilities and promoting secure coding practices.

C. USE CASE 3: PERFORMANCE OPTIMIZATION ANALYSIS

1) REQUIREMENTS

- Identify and optimize performance bottlenecks in the code.
- Analyze computational complexity and memory usage.
- Improve application responsiveness and resource efficiency.

2) RECOMMENDED TOOLS

- **Radon**: This tool focuses on code complexity metrics, such as Cyclomatic Complexity and Maintainability Index, which are important for identifying areas that need performance optimization.
- **Wily**: Wily monitors and reports various code complexity metrics over time, offering valuable insights into performance trends and potential bottlenecks.

D. USE CASE 4: DOCUMENTATION AND COMMENTS ANALYSIS

1) REQUIREMENTS

- Ensure clear and comprehensive inline documentation
- Assess the presence and quality of comments within the code.
- Improve code understandability for future maintenance and collaboration.

2) RECOMMENDED TOOLS

- **Pygount**: Pygount evaluates the amount of code comments and provides metrics on documentation quality, making it an effective tool for ensuring thorough inline documentation.

E. USE CASE 5: COMPLIANCE AND STANDARDS CHECKING

1) REQUIREMENTS

- Ensure compliance with industry-specific standards and regulations.
- Adhere to internal coding guidelines.
- Automatically check for violations of standards.

2) RECOMMENDED TOOLS

- **SonarQube**: SonarQube is recognized for its strong compliance checking features and support for various industry standards, making it suitable for projects that must meet specific regulatory requirements.
- **PyLint**: PyLint enforces PEP 8 and additional coding standards, which aids in keeping the codebase consistent and compliant.

IX. CONCLUSION

This evaluation of Python static code analysis tools follows the principles of Findability, Accessibility, Interoperability, and Reusability (FAIR). In addition to their technical prowess, the historical context of these tools makes them seminal, but they also had to pass this essential scent test as well.

The results of this study can serve as a guide for developers and organizations that aim to increase software quality, reliability and maintainability. This study promotes a sustainable developmental ecosystem in the software development domain with a clear focus on transparency, collaboration and sustainability. Our results demonstrate that comprehensive documentation, explicit performance metrics, and verification of testing are key features improving the usability and reproducibility of software. This research therefore extends the FAIR principles to provide a framework for evaluation that enables all stakeholders to make decisions informed by evidence.

By embracing a FAIR-driven approach, the software engineering community can cultivate a culture centered on transparency, collaboration, and ongoing quality improvement. Continuous research and teamwork will help refine evaluation methods, allowing for more effective responses to the ever-changing challenges in software engineering. The goal is to create a future where software development is marked by rigor, efficiency, and excellence.

This study holds practical significance as it bridges theoretical evaluation frameworks with actionable insights for developers, researchers, and organizations. By connecting Python static code analysis tools with FAIR principles, this research helps software developers make smarter choices when picking the right tools. This connection improves reproducibility, usability, and long-term sustainability. For example, in industries like IoT and healthcare—where security and quality are extremely important—these tools can help ensure compliance with standards and reduce risks. This study aims to make decision-making simpler and promote transparency, ultimately enhancing software quality and security across different sectors. For the future, creating a user-friendly web application would allow developers to easily choose and evaluate static code analysis tools based on their specific programming languages and analysis needs.

REFERENCES

[1] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J. W. Boiten, L. B. da Silva Santos, P. E. Bourne, and J. Bouwman, "The FAIR guiding principles for scientific data management and stewardship," *Sci. Data*, vol. 3, no. 1, pp. 1–9, Mar. 2016.

[2] A. Lamprecht, L. Garcia, M. Kuzak, C. Martinez, R. Arcila, E. M. Del Pico, V. D. Del Angel, S. van de Sandt, J. Ison, P. Martinez, P. McQuilton, A. Valencia, J. Harrow, F. Psomopoulos, J. Gelpi, N. C. Hong, C. Goble, and S. Capella-Gutierrez, "Towards fair principles for research software," *Data Sci.*, vol. 3, no. 1, pp. 37–59, 2020.

[3] M. Barker, N. P. Chue Hong, D. S. Katz, A.-L. Lamprecht, C. Martinez-Ortiz, F. Psomopoulos, J. Harrow, L. J. Castro, M. Gruenpeter, P. A. Martinez, and T. Honeyman, "Introducing the FAIR principles for research software," *Sci. Data*, vol. 9, no. 1, p. 622, Oct. 2022.

[4] P. A. Soha, "Reinterpretation of fair guidelines for program slicing tools," in *Proc. Int. Conf. Appl. Informat.*, 2023, pp. 1–9.

[5] E. M. del Pico, J. L. Gelpi, and S. Capella-Gutiérrez, "Fairsoft—A practical implementation of FAIR principles for research software," *Bioinformatics*, vol. 40, no. 8, pp. 1–9, Jul. 2024.

[6] P. Ferrara, A. K. Mandal, A. Cortesi, and F. Spoto, "Static analysis for discovering IoT vulnerabilities," *Int. J. Softw. Tools Technol. Transf.*, vol. 23, no. 1, pp. 71–88, Feb. 2021.

[7] V. Sachidananda, S. Bhairav, and Y. Elovici, "OVER: Overhauling vulnerability detection for iot through an adaptable and automated static analysis framework," in *Proc. 35th Annu. ACM Symp. Appl. Comput.*, Mar. 2020, pp. 729–738.

[8] D. Nikolic, D. Stefanovic, D. Dakic, S. Sladojevic, and S. Ristic, "Analysis of the tools for static code analysis," in *Proc. 20th Int. Symp. INFOTEH-JAHORINA (INFOTEH)*, Mar. 2021, pp. 1–6.

[9] Z. P. Reynolds, A. B. Jayanth, U. Koc, A. A. Porter, R. R. Raje, and J. H. Hill, "Identifying and documenting false positive patterns generated by static code analysis tools," in *Proc. IEEE/ACM 4th Int. Workshop Softw. Eng. Res. Ind. Pract. (SER&IP)*, May 2017, pp. 55–61.

[10] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Softw. Eng.*, vol. 25, no. 2, pp. 1419–1457, Mar. 2020.

[11] Q. I. Sarhan, B. S. Ahmed, M. Bures, and K. Z. Zamli, "Software module clustering: An in-depth literature analysis," *IEEE Trans. Softw. Eng.*, vol. 48, no. 6, pp. 1905–1928, Jun. 2022.

[12] I. Ruiz-Rube, T. Person, J. M. Dodero, J. M. Mota, and J. M. Sánchez-Jara, "Applying static code analysis for domain-specific languages," *Softw. Syst. Model.*, vol. 19, no. 1, pp. 95–110, Jan. 2020.

[13] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav, "A survey of static analysis methods for identifying security vulnerabilities in software systems," *IBM Syst. J.*, vol. 46, no. 2, pp. 265–288, 2007.

[14] H. Aslanyan, Z. Gevorgyan, R. Mkoyan, H. Movsisyan, V. Sahakyan, and S. Sargsyan, "Static analysis methods for memory leak detection: A survey," in *Proc. Ivannikov Memorial Workshop (IVMEM)*, Sep. 2022, pp. 1–6.

[15] D. Giebas and R. Wojszczyk, "Detection of concurrency errors in multithreaded applications based on static source code analysis," *IEEE Access*, vol. 9, pp. 61298–61323, 2021.

[16] M. Farhadi, H. Haddad, and H. Shahriar, "Compliance checking of open source EHR applications for HIPAA and ONC security and privacy requirements," in *Proc. IEEE 43rd Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 1, Jul. 2019, pp. 704–713.

[17] C. Marantos, K. Salapas, L. Papadopoulos, and D. Soudris, "A flexible tool for estimating applications performance and energy consumption through static analysis," *Social Netw. Comput. Sci.*, vol. 2, no. 1, pp. 1–11, Feb. 2021.

[18] N. Imtiaz, S. Thorn, and L. Williams, "A comparative study of vulnerability reporting by software composition analysis tools," in *Proc. 15th ACM / IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Oct. 2021, pp. 1–11.

[19] J. Zhai, X. Xu, Y. Shi, G. Tao, M. Pan, S. Ma, L. Xu, W. Zhang, L. Tan, and X. Zhang, "CPC: Automatically classifying and propagating natural language comments via program analysis," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. (ICSE)*, Oct. 2020, pp. 1359–1371.

[20] A. Kaur and R. Nayyar, "A comparative study of static code analysis tools for vulnerability detection in C/C++ and Java source code," *Procedia Comput. Sci.*, vol. 171, pp. 2023–2029, Jan. 2020.

[21] Z. Guo, T. Tan, S. Liu, X. Liu, W. Lai, Y. Yang, Y. Li, L. Chen, W. Dong, and Y. Zhou, "Mitigating false positive static analysis warnings: Progress, challenges, and opportunities," *IEEE Trans. Softw. Eng.*, vol. 49, no. 12, pp. 5154–5188, Dec. 2023.

[22] V. Bushong, D. Das, A. Al Maruf, and T. Cerny, "Using static analysis to address microservice architecture reconstruction," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2021, pp. 1199–1201.

[23] E. Amer and I. Zelinka, "A dynamic windows malware detection and prediction method based on contextual understanding of API call sequence," *Comput. Secur.*, vol. 92, May 2020, Art. no. 101760.

[24] Z. Zuo, K. Wang, A. Hussain, A. A. Sani, Y. Zhang, S. Lu, W. Dou, L. Wang, X. Li, C. Wang, and G. H. Xu, "Systemizing interprocedural static analysis of large-scale systems code with graspan," *ACM Trans. Comput. Syst.*, vol. 38, nos. 1–2, pp. 1–39, May 2020.

[25] Landscapeio. (2024). *Landscapeio/prospector: Inspects Python Source Files and Provides Information About Type and Location of Classes, Methods Etc*. Accessed: Mar. 10, 2024. [Online]. Available: https://github.com/landscapeio/prospector

[26] Pylint-Dev. (2024). *Pylint-dev/pylint: It's Not Just a Linter That Annoys You*. Accessed: Mar. 10, 2024. [Online]. Available: https://github.com/pylint-dev/pylint

[27] Roskakori. (2024). *Roskakori/pygount: Count Lines of Code for Hundreds of Languages Using Pygments*. Accessed: Mar. 11, 2024. [Online]. Available: https://github.com/roskakori/pygount

[28] Pygments. (2024). *Pygments/pygments: Pygments is a Generic Syntax Highlighter Written in Python*. Accessed: Mar. 14, 2024. [Online]. Available: https://github.com/pygments/pygments

[29] SonarSource. (2024). *Sonarsource/sonar-python: : snake: Sonarqube Python Plugin*. Accessed: Mar. 11, 2024. [Online]. Available: https://github.com/SonarSource/sonar-python

[30] Rubik. (2024). *Rubik/radon: Various Code Metrics for Python Code*. Accessed: Mar. 14, 2024. [Online]. Available: https://github.com/rubik/radon

[31] Tartley. (2024). *Tartley/colorama: Simple Cross-platform Colored Terminal Text in Python*. Accessed: March 9, 2024. [Online]. Available: https://github.com/tartley/colorama

[32] PyCQA. (2024). *Pycqa/mccabe: Mccabe Complexity Checker for Python*. Accessed: Mar. 13, 2024. [Online]. Available: https://github.com/PyCQA/mccabe

[33] Infoxchange. (2024). *Infoxchange/pylint-mccabe: Mccabe Complexity Checker As a Pylint Plugin*. Accessed: Mar. 14, 2024. [Online]. Available: https://github.com/infoxchange/pylint-mccabe

[34] Zgoda-Mobica. (2024). *Zgoda-mobica/decomplexator: Python Code Complexity Report*. Accessed: Mar. 14, 2024. [Online]. Available: https://github.com/zgoda-mobica/decomplexator

[35] Melevir. (2024). *Melevir/cognitive_complexity: Library to Calculate Python Functions Cognitive Complexity Via Code*. Accessed: Mar. 11, 2024. [Online]. Available: https://github.com/Melevir/cognitive_complexity

[36] SemanticBeeng. (2024). *Semanticbeeng/pyan: Pyan is a Python Module That Performs Static Analysis of Python Code to Determine a Call Dependency Graph Between Functions and Methods. This is Different From Running the Code and Seeing Which Functions Are Called and How Often*. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/SemanticBeeng/pyan

[37] Terryyin. (2024). *Terryyin/lizard: A Simple Code Complexity Analyser Without Caring About the C/c++ Header Files or Java Imports, Supports Most of the Popular Languages*. Accessed: Mar. 7, 2024. [Online]. Available: https://github.com/terryyin/lizard

[38] Tonybaloney. (2024). *Tonybaloney/wily: A Python Application for Tracking, Reporting on Timing and Complexity in Python Code*. Accessed: Mar. 14, 2024. [Online]. Available: https://github.com/tonybaloney/wily

[39] Jffm. (2024). *PyAnalyzer is an Extensible Python Analyzer. PyAnalyzer Performs Source Code Analysis for Python Code Based on the Standard Python 'compiler' Package. Acknowledgement: EU Funded FP6 QualOSS Project (Grant Agreement Number 033547)*. Accessed: Mar. 7, 2024. [Online]. Available: https://github.com/jffm/pyanalyzer

[40] Elmotec. (2024). *Elmotec/codemetrics: Multi Language Library for Pandas Notebook to Mine Git and Gain Insights on Your Code Base*. Accessed: Apr. 8, 2024. [Online]. Available: https://github.com/elmotec/codemetrics

[41] Rgalanakis. (2024). *Rgalanakis/pynocle: Code Metrics for Python Code*. Accessed: Mar. 9, 2024. [Online]. Available: https://github.com/rgalanakis/pynocle

[42] Priv-Kweihmann. (2024). *Priv-kweihmann/multimetric: Calculate Code Metrics in Various Languages*. Accessed: Mar. 14, 2024. [Online]. Available: https://github.com/priv-kweihmann/multimetric

HASSAN BAPEER HASSAN received the B.Sc. degree in computer science from the University of Duhok, Iraq, in 2010, and the M.Sc. degree in web applications and services from the University of Leicester, U.K., in 2015. He is currently a Lecturer with the College of Medicine, Medical Education Development Program, University of Duhok. His research interests include artificial intelligence, the Internet of Things, and programming languages.

QUSAY IDREES SARHAN received the B.Sc. degree in software engineering from the University of Mosul, Iraq, in 2007, the M.Tech. degree in software engineering (scholarship) from Jawaharlal Nehru Technological University, India, in 2011, and the Ph.D. degree in software engineering (scholarship) from the University of Szeged, Hungary, in 2023. Currently, he is an Assistant Professor with the University of Duhok, Iraq. His research interests include software engineering (static and dynamic program analysis, software testing, and software quality), the Internet of Things (data analysis, embedded systems, and IoT applications), and artificial intelligence (datasets, algorithms, and AI/ML applications). He has over 35 publications in national and international publication venues and is regularly invited to serve as a reviewer for specialized conferences and journals.

ÁRPÁD BESZÉDES received the Ph.D. degree in computer science from the University of Szeged, in 2005. He is currently an Associate Professor with the University of Szeged. His active research interest includes static and dynamic program analysis, with a special emphasis on software testing and debugging applications. He has over 100 publications and is regularly invited to serve in the program committees of various software engineering conferences and as a reviewer and an editor for software engineering and computer science journals.