Software Code Standard Improvement Automation

Kaustubh Hemang Pandya
Department of Computer Science
Bishop's University
Sherbrooke, Canada
kpandya22@ubishop.ca

Nikunjkumar Khandar
Department of Computer Science
Bishop's University
Sherbrooke, Canada
nkhandar23@ubishop.ca

Priyank Nilesh Dabhi
Department of Computer Science
Bishop's University
Sherbrooke, Canada
pdabhi23@ubishop.ca

***Abstract*—Software developers often employ static analysis tools to detect potential faults, vulnerabilities, code smells, and to evaluate code adherence to standards and guidelines early in the development process. The integration of these tools into Continuous Integration (CI) pipelines has been advocated by both researchers and practitioners. This study examines the utilization of static analysis tools in open-source projects on GitHub/Bitbucket, utilizing Jenkins CI for continuous integration. Specifically, the research investigates: (i) the tools being used and their configurations within the CI, (ii) the types of issues causing build failures or warnings, and (iii) how and when broken builds and warnings are addressed. The findings reveal that build failures in the analyzed projects primarily stem from deviations from coding standards, with some attention given to missing licenses. Failures related to potential bugs or vulnerabilities identified by tools occur less frequently, with some tools being configured to operate in a less strict mode, avoiding build failures. Additionally, the study shows that build failures due to static analysis tools are promptly remedied by addressing the underlying issues rather than simply bypassing them.**

Keywords—Continuous Integration; Static Analysis Tools; Empirical Study; Open-Source Projects

I. INTRODUCTION

In recent years, the fusion of static code analysis tools with Continuous Integration (CI) pipelines, particularly within the Jenkins automation framework, has become pivotal in modern software development. This amalgamation serves a dual purpose: ensuring code quality and adherence to coding standards, while also enabling the early detection and mitigation of potential faults and vulnerabilities. The integration of static code analysis tools within Jenkins pipelines represents a proactive approach to software development, empowering developers to identify and rectify issues at the nascent stages of the development lifecycle.

This paper delves into the utilization of static code analysis tools within Jenkins pipelines in the context of open source Java projects hosted on GitHub/Bitbucket. By focusing on the intersection of static code analysis and Jenkins-based CI, this study endeavors to provide valuable insights into the practical implementation and efficacy of such tools in real-world software development scenarios. The research seeks to address pivotal questions concerning the selection, configuration, and impact of static code analysis tools on the development process within the realm of open source projects.

Understanding how open source projects harness static code analysis tools within Jenkins pipelines holds immense significance, offering illumination on best practices in software development and furnishing invaluable lessons and insights for researchers and practitioners alike. By scrutinizing a diverse array of projects and analyzing their utilization of static code analysis tools within Jenkins pipelines, this study aims to contribute to the ongoing discourse surrounding software quality assurance and continuous integration practices in the open source community.

Through empirical investigation and meticulous analysis, this paper endeavors to uncover recurring patterns, prevalent challenges, and key success factors associated with the integration of static code analysis tools within Jenkins pipelines in open source projects. Ultimately, the findings of this research endeavor to inform future development practices and catalyze advancements in software engineering methodologies within the milieu of open collaboration and community-driven software development.

II. EMPIRICAL STUDY DEFINITION AND PLANNING

The integration of static analysis tools into Continuous Integration (CI) pipelines represents a pivotal practice in modern software development, endorsed by both researchers and practitioners. This section outlines the methodology employed in conducting an empirical study to investigate the utilization of static analysis tools within open-source projects hosted on GitHub/Bitbucket, utilizing Jenkins CI for continuous integration.

A. Research Objectives:

The primary objectives of the empirical study are as follows:

i. To identify the static analysis tools employed within the CI pipelines of open-source projects.

ii. To analyze the configurations of these tools within the CI environment.

iii. To assess the types of issues causing build failures or warnings.

iv. To examine the strategies employed in addressing broken builds and warnings.

B. Dataset Selection:

The selection of projects for analysis follows a systematic approach:

i. Identification of projects on GitHub/Bitbucket utilizing Jenkins CI through data extraction from relevant datasets.

ii. Ranking of projects based on popularity metrics obtained via GitHub APIs.

iii. Filtering out projects not incorporating static analysis tools within their CI pipelines.

C. Data Collection:

Data collection encompasses the following steps:

i. Extraction of project metadata, including GitHub/Bitbucket repository names, build framework, and popularity metrics.

ii. Retrieval of build configurations and static analysis tool settings from project repositories.

iii. Compilation of information on build failures, warnings, and their resolutions from CI pipeline logs.

D. Data Analysis:

The collected data will be subjected to comprehensive analysis, focusing on:

i. Identification of prevalent static analysis tools and their configurations within CI pipelines.

ii. Analysis of the types of issues leading to build failures or warnings.

iii. Examination of the effectiveness of strategies employed in resolving broken builds and warnings.

E. Ethical Considerations:

The study will adhere to ethical guidelines, ensuring the anonymity of project contributors and compliance with data privacy regulations.

F. Limitations:

The study may be subject to certain limitations, including selection bias in project sampling and variability in data quality across projects.

G. Conclusions:

This section delineates the methodology and planning for an empirical study aimed at elucidating the utilization of static analysis tools within Jenkins CI pipelines in open-source projects. The subsequent sections will present the findings and analysis derived from the empirical investigation.

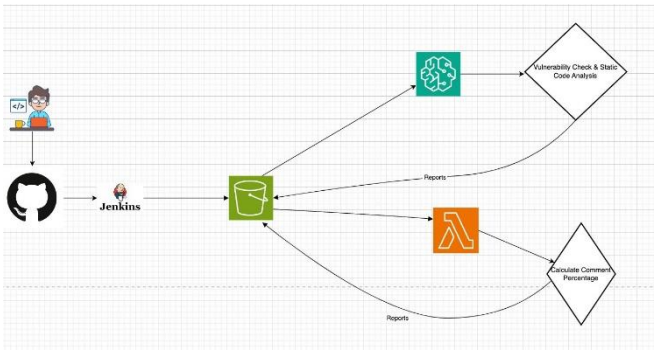

Fig. 1. Workflow

III. EMPIRICAL STUDY RESULTS

This section presents the findings derived from the empirical investigation into the utilization of static analysis tools within Jenkins CI pipelines.

A. Static Analysis Tool Usage:

The analysis revealed a diverse array of static analysis tools being employed within Jenkins CI pipelines. Commonly utilized tools include Checkstyle, autopep8, pyflakes, cpplint, pylint, bandit, safety and pyupgrade. These tools were configured to analyze various aspects of code quality, including style adherence, potential bugs, code smells, and security vulnerabilities.

a. autopep8: This module is used for automatically formatting Python code to adhere to the PEP 8 style guide.

b. pyflakes.api: This module is used for static analysis of Python code to detect errors or issues.

c. checkstyle: This variable stores the output generated by the Checkstyle tool when checking Java code. The function `check_java_code(file_path)` invokes the Checkstyle tool with the specified configuration file (`sun_checks.xml`) to analyze the Java code. The Checkstyle tool examines the Java code against predefined coding conventions and generates a report highlighting any violations. The output stored in `checkstyle_output` is later processed and included in the final results uploaded to S3.

d. cpplint: This variable contains the output produced by the CPPLint tool when analyzing C++ code. The function `check_cpp_code(file_path)` executes CPPLint on the provided C++ file to identify any style violations or errors. CPPLint examines the C++ code and reports issues related to code formatting, naming conventions, and potential errors. The output stored in `cpplint_output` is subsequently processed and incorporated into the final results sent to S3.

e. pylint: This function parses the output generated by the Pylint tool when checking Python code. The function `check_python_code(file_path)` executes Pylint on the specified Python file, capturing its output. Pylint analyzes the Python code for style violations, errors, and conventions, and generates a structured JSON output. `parse_pylint_output()` processes the JSON output from Pylint, extracting relevant information such as file paths, line numbers, messages, and symbols associated with detected issues. The parsed output is then returned as a list of dictionaries representing pylint issues, which is further included in the final results uploaded to S3.

f. Bandit: Bandit is a security-focused static analysis tool for Python code. It scans Python scripts and detects common security vulnerabilities and issues, such as potential injection vulnerabilities, insecure cryptographic practices, and hardcoded passwords.

g. Safety: Safety is a tool that checks Python dependencies for known security vulnerabilities. It analyzes the dependencies listed in a Python project's requirements.txt file or installed packages and compares them against the National Vulnerability Database (NVD) to identify any vulnerabilities that may exist in the dependencies.

h. Pyupgrade: Pyupgrade is a tool for upgrading Python code to newer syntax versions. It automatically applies safe and compatible transformations to Python code, helping to modernize

B. Configuration Within CI Pipelines:

Static analysis tools were typically integrated into the CI pipeline as automated steps executed during the build process. Configuration settings varied across projects, with developers customizing tool configurations to suit project-specific requirements. Notably, some tools were configured to operate in a less strict mode, allowing for the detection of issues without causing build failures.

C. Types of Issues Causing Build Failures or Warnings:

The primary issues leading to build failures were deviations from coding standards detected by tools like Checkstyle and PMD. These deviations encompassed violations of formatting conventions, naming conventions, and other stylistic guidelines. Additionally, some projects exhibited build failures due to missing licenses, detected by license compliance tools.

D. Strategies for Addressing Broken Builds and Warnings:

The study observed prompt resolution of build failures and warnings within the analyzed projects. Developers actively addressed underlying issues identified by static analysis tools, leading to swift remediation of broken build and warnings. Strategies for resolution included code refactoring, adjusting tool configurations, and addressing licensing issues.

E. Implications and Insights:

The findings underscore the importance of integrating static analysis tools into CI pipelines for early detection and mitigation of code quality issues. By promptly addressing detected issues, developers ensure the maintenance of high-quality code standards and enhance the overall stability and reliability of software projects.

F. Limitations and Future Research Directions:

It is essential to acknowledge potential limitations, including the representativeness of the sampled projects and the inherent variability in tool configurations and project contexts. Future research may explore the effectiveness of specific tool configurations in mitigating different types of code quality issues and investigate the impact of static analysis tool adoption on overall project quality metrics.

G. Conclusion:

This section elucidates the empirical findings regarding the utilization of static analysis tools within Jenkins CI pipelines in open-source Java projects. The insights gleaned from this study contribute to the ongoing discourse surrounding software quality assurance practices and inform best practices for integrating static analysis tools into CI workflows.

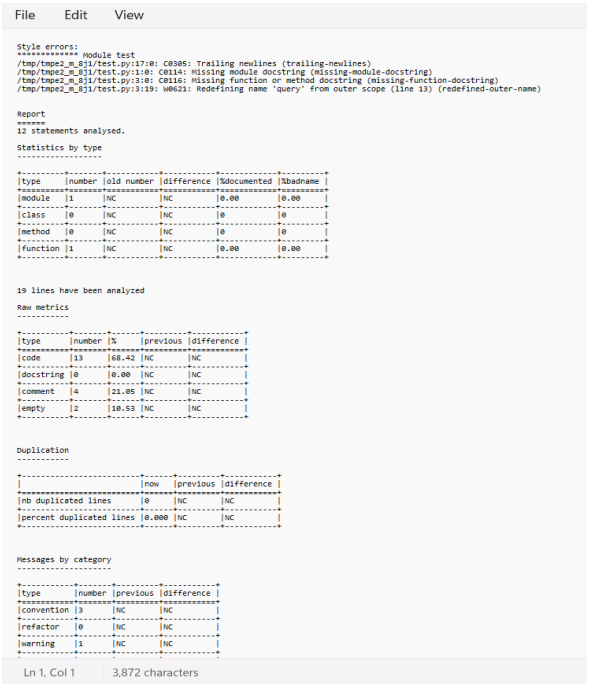

Fig. 2. suggestions.txt

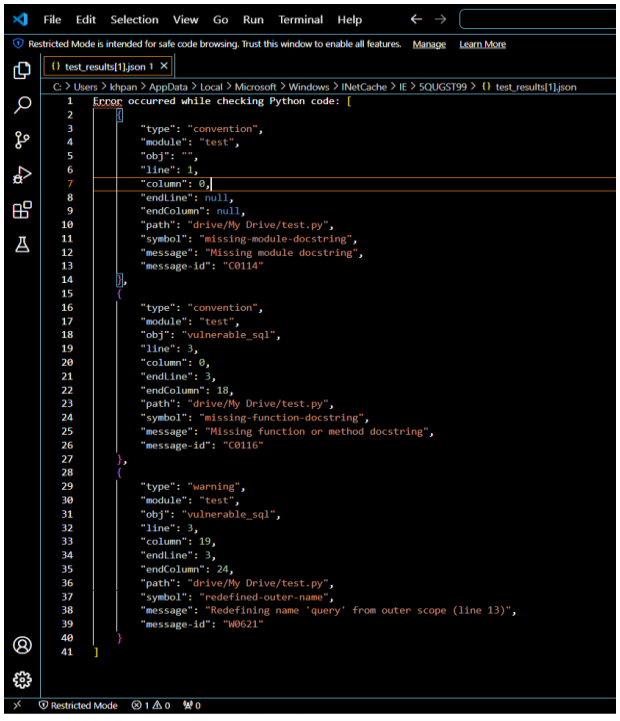

Fig. 3. results.json

Fig. 4. static_analysis_output.txt

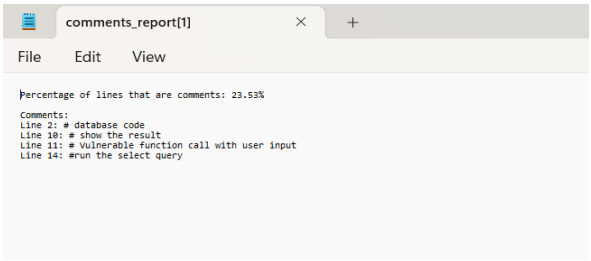

Fig. 5. comments_report.txt

IV. THREATS TO VALIDITY

This section outlines potential threats to the validity of the empirical study conducted on the utilization of static analysis tools within Jenkins CI pipelines in open-source Java projects.

A. *External Validity:*

1. Sampling Bias: The selection of projects from GitHub/Bitbucket may introduce bias, as it may not represent the entire spectrum of open-source Java projects. Projects hosted on other platforms or utilizing different CI infrastructures may exhibit different practices.
2. Generalizability: The findings may not be generalizable to all open-source Java projects, as the sampled projects may not be representative of the broader population of projects.

B. *Internal Validity:*

1. Tool Configuration Variability: Variability in how static analysis tools are configured within CI pipelines may impact the consistency and comparability of results across projects.
2. Data Collection Errors: Inaccuracies or inconsistencies in data collection, such as misinterpretation of build logs or incomplete extraction of configuration settings, may affect the reliability of the findings.

C. *Construct Validity:*

1. Measurement Bias: The metrics used to assess the effectiveness of static analysis tools, such as the number of build failures or the severity of detected issues, may not fully capture the true impact of these tools on code quality.
2. Tool Selection Bias: The choice of static analysis tools included in the study may bias the results, as other tools with different capabilities or configurations may yield different findings.

D. *Conclusion Validity:*

1. Inference Errors: The conclusions drawn from the empirical findings may be subject to interpretation bias or oversimplification, leading to inaccurate or incomplete insights.
2. Temporal Validity: The study's findings may become outdated over time, as software development practices and tooling evolve, potentially rendering the conclusions less applicable in future contexts.

E. *Mitigation Strategies:*

To address these threats to validity, efforts were made to ensure a diverse selection of projects, rigorous data collection and analysis procedures, and transparent reporting of findings and limitations. Additionally, future research could aim to replicate the study with a larger and more diverse sample of projects and employ robust statistical methods to enhance the reliability and generalizability of the findings.

V. RELATED WORK

This section discusses related work on Continuous Integration and on the use of ASCATs in industrial and open-source projects.

A. *Continuous Integration and Build Failures*

In recent years, researchers have studied the CI practices adopted in industry and open source. As reported by St˚ahl, CI is becoming increasing popular in software development. This is confirmed by Hilton *et al.*, who conducted an extensively study about the usage of CI infrastructure in open-source projects showing how this practice has become increasingly popular in OSS. Instead, as regards the adoption of CI in industry, Laukkanen noticed how developers, adopting CI process, deal with several technical and social challenges like the test infrastructure. Moreover, St˚ahl

showed that in industry there is not a unique and homogeneous CI practice. Vassallo investigated the adoption of CI in a large financial organization by surveying 158 DevOps. Among other results, they indicated how ASCATs within CI are mainly used to get rid of unused/unreachable code. This is consistent with some of the checks we found highly used like Check Style's *Imports*.

Previous researchers have also investigated on the nature and impact of build failures (in CI and not). Miller reported on a study conducted at Microsoft and showed that builds mainly fail for static analysis (40%), unit testing, compilation and server failures. In our study, the observed percentage of broken builds due to ASCATs is much smaller than what observed by Miller. Differently than Miller, we perform a deep investigation on the nature of build failures due to ASCATs, other than looking at their percentage and their removal. Studies conducted by Seo Beller et al. [14] focused on compilation and testing build failures, respectively. Kerzazi measured the impact of build failures in a software company, analyzing more than 3k builds, and tried to identify possible causes and effects interviewing 28 software engineers.

REFERENCES

[1] "Apache Maven. http://maven.apache.org/ (last access: 02/10/2017)."

[2] "Apache-rat. https://creadur.apache.org/rat/ (last access: 02/10/2017)."

[3] "CheckStyle. http://checkstyle.sourceforge.net/ (last access:
02/10/2017)."

[4] "Clirr. http://www.mojohaus.org/clirr-maven-plugin (last access:
02/10/2017)."

[5] "FindBugs. http://findbugs.sourceforge.net/ (last access: 02/10/2017)."

[6] "Gradle. https://gradle.org/ (last access: 02/10/2017)."

[7] "jDepend. http://www.mojohaus.org/jdepend-maven-plugin/ (last access:
02/10/2017)."

[8] "License-gradle-plugin. https://github.com/hierynomus/ license-gradle-plugin (last access: 02/10/2017)."

[9] "PMD. https://pmd.github.io/ (last access: 02/10/2017)."

[10] N. Ayewah and W. Pugh, "The google findbugs fixit," in Proceedings of the 19th international symposium on Software testing and analysis. ACM, 2010, pp. 241–252.

[11] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in Proceedings of the 2013 international conference on software engineering. IEEE Press, 2013, pp. 712–721.

[12] K. Beck, Extreme programming explained: embrace change. Addison- Wesley Professional, 2000.

[13] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open-source software," in 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1. IEEE, 2016, pp. 470–481.

[14] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An analysis of travis CI builds with GitHub," PeerJ PrePrints, vol. 4, 2016. [Online]. Available: http://dx.doi.org/10.7287/peerj.preprints.1984v1

[15] ——, "Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in Proceedings of the 14th working conference on mining software repositories, 2017.

[16] G. Booch, Object Oriented Design: With Applications. Benjamin Cummings, 1991.

[17] W. J. Conover, Practical Nonparametric Statistics, 3rd ed. Wiley, 1998.

[18] C. Couto, J. E. Montandon, C. Silva, and M. T. Valente, "Static correspondence and correlation between field defects and warnings reported by a bug finding tool," Software Quality Journal, vol. 21, pp.241–257, 2011.

[19] P. Duvall, S. M. Matyas, and A. Glover, Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series). Addison-Wesley Professional, 2007.

[20] R. J. Grissom and J. J. Kim, Effect sizes for research: A broad practical approach, 2nd ed. Lawrence Earlbaum Associates, 2005.