Towards Streamlining Auditing for Compliance with Requirements in Open-source Software at NASA

1st Carlos Paradis
KBR @ NASA Ames Research Center
Moffett Field, United States
ORCID: 0000-0002-3062-7547

2nd Ivan Perez
KBR @ NASA Ames Research Center
Moffett Field, United States
ivan.perezdominguez@nasa.gov

3rd Misty Davies
NASA Ames Research Center
Moffett Field, United States
misty.d.davies@nasa.gov

***Abstract*—Context: Software that operates in critical environments must be developed and maintained following strict software engineering and development processes. The instantiation of such processes may vary per project; however, once decided upon, projects must undergo audits to evaluate compliance with such requirements.**

Aim: We propose that audit effort can be reduced when requirements are realized by leveraging commonly used open-source infrastructure for version control, issue tracking and continuous integration, and the generated records are analyzed using a repository mining software tool to quantify process compliance.

Method: We perform a case study in the NASA-funded Copilot project, utilizing Kaiaulu, a repository mining software tool. We define four software compliance metrics based on the Copilot's requirements, and analyze their impact on source code quality.

Results: Our work demonstrates how it is possible to leverage existing open source tools and platforms to facilitate software certification and qualification, and to streamline the auditing process required even when stringent requirements must be enforced.

Conclusion: Together, both project and tool can be utilized to visualize project compliance, and metrics can be defined to more easily identify process irregularities to minimize auditing efforts.

Project Repository: github.com/Copilot-Language/copilot

Tool Repository: github.com/sailuh/kaiaulu

***Index Terms*—mining-software-repositories, software-process, code-quality**

I. Introduction

Software engineering practitioners often conduct quality auditing of the development process to assure conformance with organizational standards. Standards to follow may be internal to the organization, as well as public, and they vary depending on the domain and criticality of the project (e.g., DO-178C [1] in aviation, IEC 62304 [2] in medicine).

For example, NASA requires all software to comply with an extensive list of requirements governing its development, maintenance, operation, management and retirement, as outlined in the "NASA Software Engineering Requirements" (NASA Procedural Requirements (NPR) 7150.2).[1] The specific requirements that a NASA project must comply with are determined based on the level of criticality of the software, which is partly based on when and how it is going to be used, the risk that a software failure could carry with it, and how that could affect a mission where the software is intended to be used.[2]

NPR 7150.2's requirements are intentionally high-level, and it is up to each development team to determine how they will comply with them, and to demonstrate compliance. To justify that the requirements will be met, teams must provide a number of software planning documents covering aspects such as architectural design, configuration management and testing, among many others. Such plans only comprise the "easier" half of the evidence required: the planning, development, maintenance and operation of the software should also be auditable at different points during the project's lifecycle and by different people, to confirm that the process outlined in the documents is being followed.

Assuring that the process defined is actually followed by the development team is usually difficult [3]. Auditing that the process has been followed throughout the life of a project can be very costly, especially as projects grow in size, impact, longevity and personnel. The incorporation of new team members who may not have been trained in the specifics of the software engineering process, as well as changing requirements and new features needed in the project, pose a serious challenge in terms of compliance and in terms of auditing for compliance. Oftentimes, the evidence for these audits is manually compiled, which can take months or longer even for relatively simple projects.

This paper[3] discusses how we can leverage an existing open-source tool, Kaiaulu [4], to obtain evidence needed to support compliance with NASA's NPR7150.2. We demonstrate our approach with a case study on the NASA-funded, open-source runtime verification framework Copilot.[4] The case of Copilot is specifically interesting due to its open nature: not only is the software open source, but the implementation of the requirements of NPR7150.2 leverages existing infrastructure often used by the open-source community, such as Git, GitHub and Travis CI. Additionally, a substantial amount of evidence of Copilot's compliance with the requirements is documented publicly in the project's public issue tracker

[1]https://nodis3.gsfc.nasa.gov/displayDir.cfm?Internal_ID=N_PR_7150_002D_&page_name=main

[2]https://nodis3.gsfc.nasa.gov/displayDir.cfm?Internal_ID=N_PR_7150_002D_&page_name=AppendixC

[3]A pre-print of this manuscript can be found on NASA's institutional repository: https://ntrs.nasa.gov/citations/20230011257

[4]https://github.com/Copilot-Language/copilot/

and code repository. More specifically, our work makes the following contributions:

- We demonstrate how we can extend an existing open-source tool to detect process compliance, by measuring different aspects required by the project's development plans.
- We use Copilot as an initial use case to analyze whether compliance with specific rules, over time, correlates with software quality.

II. Background and Related Work

A. Related Work

The extensions to the tool provided in this work belong to the field of process mining [3], [5]–[8]; related work focuses either on process discovery [5], [6] or auditing [3]. The goal of process mining is to extract information about processes from transaction logs. These transaction logs can then be analyzed in different *perspectives*, such as the characterization of all possible paths in the process (process perspective), the structure of the organization in regards to people and their roles (organizational perspective), and more [7]. Our work is similar, in that we seek to audit a process, and for that we require mining process logs, but differs in that we chose to represent them as software metrics.

Both Rubin et al. [5] and Jensen et al. [6] propose a method to discover processes from open-source logs. Jensen et al. [6] propose using text and link analysis to recover processes; however, no results are presented. Rubin et al. use the ProM framework [7] on ArgoUML's version control system. The ArgoUML version control system logs are generalized using heuristics before being modeled using ProM. For example, if a modified file contains "models" in its path, then it is mapped to "DES", whereas a file ending in ".java" is mapped to "CODE". Our work defines process metrics on the source dataset without transformations, which we argue is important for providing actionable insight to stakeholders when process deviation occurs. Similar ideas have also been proposed to process and unify different software infrastructure to better understand process mining in software repositories [8], [9]. However, these works are mostly non-empirical and do not provide tools. A subset of process mining literature in software repositories has also focused on the lifecycle of issues [10], in particular bugs [11]–[13].

For process auditing, closely related to our work, Lemos et al. [3] evaluate a large Brazilian software database aiming at detecting inconsistencies between a process model and its execution log. The authors first define the companies' formal process, and then evaluate its log database using the ProM Framework [7]. Similarly, we also define in this work a subset of NASA's NPR 7150.2, and evaluate inconsistencies given the execution log. Our work, however, diverges in the process granularity, coverage, and method in a way that we believe is more accessible to the open-source community and industry alike. Specifically, the process defined in [3] identifies process steps such as 1) documentation analysis by managers, 2) planning and budgeting, 3) customer approval, and 4) technical specification validation, 5) development, 6) code verification, 7) testing, 8) and documentation.

Our formal process definition is more specific, directly verifying conformance *per commit* or *per issue*. This specificity enables, for example, immediately identifying during the audit those activities developers can understand, correct and learn from. In contrast to [3], our method of analysis implements compliance metrics, instead of models that derive the probability of process state transitions.

Lehtonen et al. [14] use visualizations to analyze software development processes, with the purpose of aiding in understanding and in improvement. Their analysis is based on the time it takes for issues to be addressed, as well as the number of issues open at a specific time, consistent with standard requirements and practices in Scrum. In contrast, we treat events as discrete and only pay attention to their order, and not the time between events. Our practices are generic enough that it is possible to adopt a specific development method (e.g., Scrum) on top. Matiila et al. [15] expand on those ideas and combine data from issues, development and usage to produce combined visualizations that seek to help identify issues with continuous delivery. The main goal of such analysis was to identify the time that issues took to be resolved or features took to be implemented. Our work focuses on trying to measure compliance with a plan in order to help identify and prevent process deviation.

Saltz [16] uses visualizations implemented in R to measure project evolution in a Kanban process. Our work is specific to Software Engineering, and it measures compliance with a specific implementation of NPR7150; thus differing in terms of process, metrics and tool.

Tools like GitHub and Jira allow for the implementation of some of these rules directly on the repository and issue tracking system. For example, GitHub supports branch protection rules that require status checks (i.e., checklists and tests) to pass before a pull request can be merged. While such mechanisms enable implementing specific rules, Kaiaulu is more versatile and is able to capture and check for a larger set of of rules and metrics. Finally, some tools have been reported in the literature to facilitate building pipelines around process mining tools [17]. We believe Kaiaulu complements these tools, by providing means to mine data for process mining models, or alternatively, as we proposed in this work, more specific metrics for software compliance.

In the next subsections, we provide background for the process NASA software should follow (NASA NPR 7150.2), the open-source project that serves as our use case study (Copilot) and how it was instantiated, and the mining software repository tool (Kaiaulu) which we extended to assess process compliance in Copilot in the form of software compliance metrics.

B. NASA NPR 7150.2

NASA software is required to comply with the requirements outlined in the "NASA Software Engineering Require-

ments" (NASA Procedural Requirements (NPR) 7150.2)[5], and grouped by level of criticality.

NPR 7150.2's requirements are intentionally high-level. For example, requirement SWE-054 of edition C of NPR7150.2 reads: "The project manager shall require the software developer(s) to provide NASA with electronic access to the source code developed for the project in a modifiable format." SWE-054 does not mandate a specific location, communication mechanism, or programming language. It merely indicates that the software developer must give the project manager access to the source code (as an example, providing the compiled object code and not the original source code would go against the requirements).

To ensure that projects comply with such requirements, they must provide two additional pieces of evidence. The first is a series of documentation and planning documents. The second is additional records produced during the stages of software development, operation, maintenance and retirement. Such records contain, for example, a list of the tests run for each version, the releases published, changes made, or the bugs found in the software, among others.

A substantial amount of effort is needed to audit the records produced during the software lifecycle and determine whether they deviate from the process proposed in the documentation and mandated by the requirements. Furthermore, such information continues to grow as the project continues to run, and the complexity of handling the information grows even further if new features must be added, or new people are added to the team. For example, one of the requirements is that "[t]he project manager shall identify, initiate corrective actions, and track until closure inconsistencies among requirements, project plans, and software products.". As changes are performed to the software, and as bugs detected and fixed, so will grow the amount of information needed to be audited. Our work, as we will see in the following, alleviates this burden by automating part of the auditing effort.

C. Copilot

For the purposes of illustrating our ideas, we use an existing NASA-funded project as a use case.

Copilot is a runtime monitoring language and framework frequently used by NASA to perform monitoring tasks on unmanned aerial vehicles. A Copilot specification lists conditions to check for, together with handling functions to execute when such conditions are met. The Copilot compiler generates C code from such specifications, which can then be integrated as part of a larger system.

Copilot was originally developed by Galois Inc and the National Institute of Aerospace under NASA funding in 2010, and released as open source under a BSD license. It has, since then, seen multiple revisions and, in 2020, the project started going through the process necessary to be used in flights of criticality level C. It was determined that Copilot itself was Class D software, and that the code produced using Copilot would have to meet the requirements for Class C.

As part of that process, the project underwent a number of changes:

- Plans were developed to comply with the requirements of NPR7150.
- A new software development process was put in place, in compliance with the documents.
- The libraries that comprise Copilot were refined, simplified and fixed.
- The toolchain was extended with a verifier, which provides a formal proof that the C code generated conforms to the language's formal semantics.

The new software development process indicates that the source code of Copilot must be stored in a git repository, and that it must be made available in a public repository on GitHub. Any changes made to the software also have to be tracked using the issues utility provided by GitHub, and that, prior to accepting a pull-request, the proposed changes have to be tested on Travis CI, leaving a log of the successful build.

In principle, the public nature of this process does not make complying with NASA's procedural requirements, or auditing for such compliance, any easier or harder than using private tools or methods. However, many open source tools and web applications, including git, GitHub and Travis CI, facilitate integration into other tools via additional libraries or REST APIs. In the rest of this paper, we demonstrate that, by following a very systematic development process and by using tools that analyze data available via such APIs, we can reduce the effort required to audit development records by several orders of magnitude.

D. Kaiaulu

To automate the evidence analysis in this work, we leverage Kaiaulu [4]. Kaiaulu is an R package that helps in the understanding of evolving software development communities, and of the artifacts (gitlog, mailing lists, files, etc.) that developers collaborate and communicate with. Our work builds on the Kaiaulu authors' prior work on analysis of GitHub Events [18] and the protocol developed in Copilot[6] to comply to NPR 7150.2 Class D Requirements on GitHub.

III. Method

In this section, we describe our methodology for process compliance. Figure 1 provides an overview of three of the four metrics defined in this section: **Status**, **Milestone** and **Travis**. Broadly, NASA's Procedural Requirements requires that process compliance is implemented in the software's life cycle. The Copilot project instantiate this requirement through various processes defined on GitHub, such as the use of change request progress using GitHub tags, the use of comments, or code branching rules. Because GitHub also tracks developer interaction with the project, Kaiaulu then automates and measures the process compliance of developer interaction in the platform.

[5]https://nodis3.gsfc.nasa.gov/displayDir.cfm?Internal_ID=N_PR_7150_002D_&page_name=main

[6]https://github.com/Copilot-Language/copilot

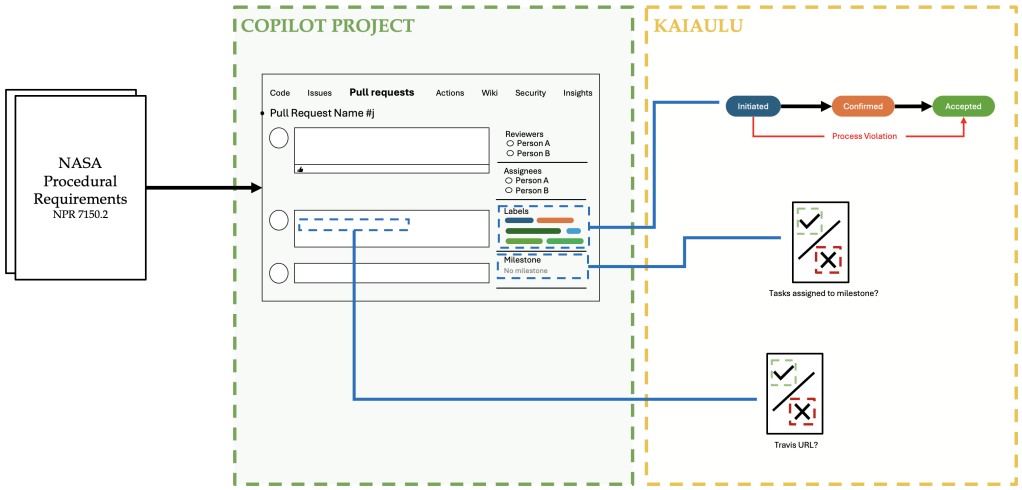


Fig. 1: Process Compliance Method

A. *Software Process Compliance Metrics*

The plans that indicate how Copilot implements the requirements in NPR7150.2 outline a specific process to guarantee compliance. Among other aspects, the process covers how change requests are to be documented, filed, planned and processed, how commits to the source code repository are to be organized, how commit messages must document changes, what kinds of tests must be performed and how, etc. For example, NPR7150.2 contains the requirement: "SWE-054: The project manager shall require the software developer(s) to provide NASA with electronic access to the source code developed for the project in a modifiable format." To meet his requirement, Copilot's Configuration Management Plan (CMP) establishes that the code will be publicly available in a repository on GitHub.

Copilot's CMP dictates a process for branch management, commit management, and change request processing that leverages existing open-source friendly infrastructure, including features available on GitHub and Travis CI. This provides an opportunity to analyze the repository and the event API of GitHub to detect if the CMP is being followed.

To demonstrate how we can automatically audit for compliance with some parts of such process, we have defined four metrics that showcase how different aspects of the NPR7150.2 can be quantified.

Status. NPR7150.2 includes several requirements related to how changes can be made to the software. SWE-080 requires that "[t]he project manager shall track and evaluate changes to software products", and SWE-054 states that "[t]he project manager shall identify, initiate corrective actions, and track until closure inconsistencies among requirements, project plans, and software products".

Copilot implements these requirements by mandating that all change requests (CRs) be identified by an issue on GitHub, and that labels be used to determine its status and the type of the change request (bug, feature, maintenance). To determine that the change request has been properly evaluated, planned, tested and completed, each issue that will be addressed by the project goes through a series of stages (in order): *Initiated* (the CR has been formally filed), *Confirmed* (it has been confirmed that the issue indeed is present, or that the requested feature is lacking), *Accepted* (the CR has been accepted in the project and it will be addressed in a future release), *Scheduled* (the CR has been assigned a person or several people to work on it, and a milestone), *Implementation* (the designated implementers are working on it), *Verification* (the solution is undergoing verification prior to being accepted in the project), and *Closed* (the issue has been completed and is now closed). On GitHub, the stage in the lifecycle of an issue is reflected via a label with prefix "`CR:Status`" (e.g., `CR:Status:Initiated`).

To determine compliance with this process, we parse the GitHub's events for every issue in Copilot, specifically, assign-label events, and check if the order of assigned labels matches the required order. This compliance metric ranges from 0 to 1, where 1 indicates a perfect match.

Milestone. Changes made to the software must follow from the project requirements, and it must be possible to check for such compliance. Specifically, requirement SWE-037 dictates that "[t]he project manager shall define and document the milestones at which the software developer(s) progress will be reviewed and audited".

With respect to the code itself, Copilot implements this requirement by leveraging the support for milestones on GitHub. Milestones denote software releases (i.e., baselines), and are tagged with their expected version number. Milestones are programmed two months apart, and issues to be addressed for a specific release are assigned the corresponding milestone after a discussion between the Technical Lead and the Project Manager.

We can partly detect compliance with this requirement by measuring whether all issues actually addressed were assigned a milestone at some point. Compliance of this metric

is binary and represented as 0 or 1, with 1 indicating that the issue is assigned a milestone.

Travis. Tests are an integral part of reliable systems at NASA. Requirements on testing at all levels (e.g., unit, integration, system) become more stringent as software increases in criticality (measured by the corresponding *class*; see "Software Classifications" under NPR7150.2D). Multiple requirements in the NPR7150.2 relate to testing. Among them, we highlight SWE-068, which requires that "[t]he project manager shall evaluate test results and record the evaluation".

Such a requirement in Copilot is met via multiple documents and processes. The test plans indicate how tests will be carried out, and indicate that a continuous integration server (Travis CI) shall be used throughout the development process to run all tests. Furthermore, when an issue is closed, the change approver must leave a comment containing a reference to a test log generated by the CI server, indicating that the test build ran correctly.

We leverage GitHub's API to obtain all comments of every issue, and perform regex matching for Travis CI URLs. If they are present, the metric is assigned value 1, otherwise 0.

Commit Issue. Requirement SWE-080 states that "[t]he project manager shall track and evaluate changes to software products", and SWE-054 states that "[t]he project manager shall identify, initiate corrective actions, and track until closure inconsistencies among requirements, project plans, and software products". In principle, access to a source code repository that keeps the history of changes is enough to understand the changes being made to the software. In practice, however, determining the impact of a change and at which point in history it can be considered completely addressed may be hard if commits are not in a consistent order, if they address multiple issues at the same time, or if a series of changes reflects the process followed by a developer to figure out a solution, rather than the process that the software should follow to incorporate said solution.

To make the implementation of SWE-080 and SWE-054 manageable, Copilot's Configuration Management Plan requires following a process inspired by Gitflow [19]. More specifically, commits are done in a development branch separately from the master branch, until changes are ready to be merged. In this manner, the master branch contains only merge commits. Furthermore, each commit message in the development branch must refer to the (only) issue being addressed by that branch, in the summary line of the commit message, and the merge commit in the master branch must also refer to the issue by adding "`Close #<issue_number>.`" at the end of the summary line. In addition to these rules, development branches must always be rebased on top of the current master prior to merging them.

We can verify some of these rules by checking that, for every merge commit, there exists a branch with at least one matching commit, as determined by their respective commit messages. If a match exists, the metric is assigned value 1, otherwise, the metric is assigned value 0.

B. Mapping Software Compliance Metrics to Source Code Files

The software compliance metrics in the previous section are defined *per issue*, or *per commit*. To better understand the correlation between software compliance and code quality, we propose to also associate these metrics to individual files. To do so, we must determine the files that each metric is associated with, and how individual results can be combined into one joint result for each metric and file. In our case, we limit ourselves to files ending in ".hs", which identify Haskell source code, the language in which Copilot is implemented.

For metrics defined *per issue*, we first establish a correspondence between files and issues. This is done indirectly via commits and commit messages. Based on the rules described before, when an issue is addressed, one or more commits must be made that indicate the issue number in the subject line of the commit message. In turn, each commit identifies a set of files. We can compose this association to find out, for each file in the tree, which issues led to changes affecting that file, making it possible to also associate any metric for an issue to a metric for files modified while addressing that issue. Since a file may relate to multiple issues, we define a file's software compliance metric as the average of all file changes to issues they are associated with for a particular metric. For the purposes of this work, we consider that a file changes each time a change to such file is *committed* to the repository. To illustrate our metric, suppose a file is associated with 3 issues, with 1, 2, and 1 file changes respectively, and its *Travis-CI Build* issue compliance metrics are $\{0,1,0\}$. The file's *Travis-CI Build metric* is thus $\frac{1*0+2*1+1*0}{1+2+1} = 0.4$. An alternative to this metric would be to simply average a file's compliance metric based on the issues, not accounting for the number of file changes in them. In that case, using the same example, the file metric would be $\frac{0+1+0}{3} = 0.33$. We chose the former definition, i.e., to weigh the file compliance metrics by the file changes, because we are interested in assessing if *file changes*, when performed on non-compliant issues, have an impact on their quality. We make no claims with respect to whether this metric is the best for all possible scenarios; we choose it in our case because it captures the information we want to obtain.

Following the same rationale, we can map the *per-commit* metric *Master and Development Branch* to files. If a merge commit does not have an associated commit in a branch with matching issue number, then all the files modified by the merge commit are assigned the metric score 0, or otherwise 1.

C. Quality Metrics

To assess the correlation of software compliance in Copilot on source code, we utilized both bug count and churn, which are are commonly used in Software Engineering literature. The number of bugs associated with a file are obtained by counting the number of issues that contain the label "`CR:Type:Bug`" that a file is associated with. Churn is derived directly from

the git log, by adding the number of lines added and removed. We use scatterplots to understand the correlation between the variables, as opposed to correlation tests, to facilitate the interpretation of the results.

Our extension in this work defines the software compliance metrics aforementioned. The capabilities to parse Git and GitHub and to visualize them as networks is available in Kaiaulu. Our work, however, is the first to highlight opportunities for software process mining using the tool and map them to files.

IV. Discussion

A. Dataset

We describe the dataset in which our compliance metrics are applied by representing it as networks. Figure 2 displays a file-issue network (collaboration), and an issue-developer network (communication). These networks can be derived from commit messages and issue tracker replies using Kaiaulu: If a commit message, which tracks a file change, contains a reference to an issue, then the file and issue nodes are connected in a graph. Similarly, if a developer replies within an issue, then a developer node and an issue node are connected in the graph.

We highlight in this figure two filtering criteria applied to the data before our compliance metrics are computed. The first filter criteria can be observed by comparing sub-figures 2a and 2c against sub-figures 2b, 2d. The second filtering criteria are the red nodes, which are a subset of the issue blue nodes in all sub-figures.

For the first filtering criteria: in section II-C we noted that Copilot began development in 2010 and underwent revisions in 2020 to obtain its Class D classification. In the scope of this work, we used the project's commit and issue log data, which are publicly available as the project is fully open source. The commits range from September 2010 to date (April 2023); its issues range from October 2021 to date (April 2023). Because we are interested in demonstrating the compliance metrics in a Class D project, we use data after 05/14/2020. Specifically, sub-figures 2a and 2b show the entire open-source Copilot ecosystem of developers, files, and issues. Sub-figures 2c and 2d showcase this ecosystem subset after 05/14/2020, which are used in this work. The communication network (developer-issue) is expected to be identical, as its dataset starts in 2021. However, we can observe Figure 2a and 2c differ in respect to isolated files (yellow nodes) in the graph. This means that, after Copilot's process changed, file changes began being consistently annotated with issues.

The second filtering criteria subsets which issues we check for compliance (i.e. the compliance metrics "Milestone", "Status", and "Travis"). The intuition here is that Copilot only began issue compliance some time after 2020. While in theory a second timestamp could be utilized to subset the dataset, older issues which were addressed after such timestamp and followed the new process would be missed. Additionally, not every issue should be verified for compliance. For example, issues that do not result in changes to the code (e.g., questions, invalid issues, and those that are never filed normatively and addressed), would fail the defined metrics. We therefore chose to only verify for compliance issues whose label contained any "CR:Status". This guaranteed the criteria above were met. The subset of issues which contained "CR:Status" are highlighted as red nodes in sub-figures 2c and 2d. While the use of "CR:Status" is specific to Copilot, other projects reusing our toolchain can leverage labels or other issue metadata to subset their dataset accordingly for compliance metrics.

Lastly, we note the graphs in this work are interactive when executed in Kaiaulu, and are labeled when zoomed in, which allow for exploration. For example, in sub-figure 2c, we can observe a few issues (blue nodes) surrounded by a large number of files (yellow nodes). Interacting with the graph reveals that the two largest nodes refer to issues #248, #316, and #88[7]. These 3 issues respectively refer to the re-structuring of Copilot repository, style guide compliance, and clean-up and merge of Copilot repositories, all of which caused changes to a large number of files. Figure 2d shows the structure collaboration in Copilot, which highlights two core developers and contributors, one of which is no longer involved in the project, and the second which is a co-author of this work. The current collaboration structure in Copilot highlights the importance of a well-defined process in order to minimize overhead when handling project contributions, and the risk to the project posed by having a large number of duties concentrated on a very small amount of developers. Further automation of the process audit to ensure compliance, therefore, helps towards minimizing the overhead of handling project contributions.

B. Compliance Metrics

As a result of the two filtering criteria, a total of 706 commit hashes were analyzed for the Commit Issue compliance metric, and 66 (red) issues out of the 411 (blue) issues for the remaining metrics. Figure 3 summarizes Copilot's compliance with respect to the metrics defined as a boxplot for each metric. The values obtained for each metric are discussed in detail in the rest of the section.

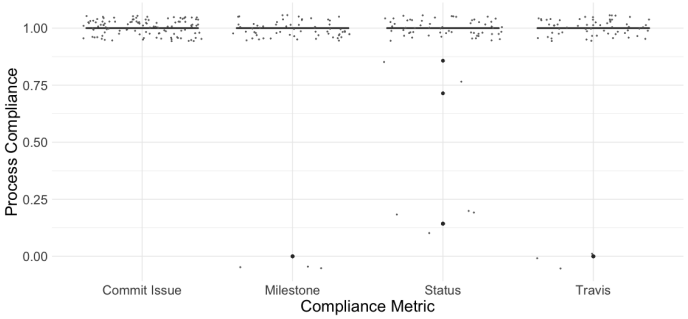
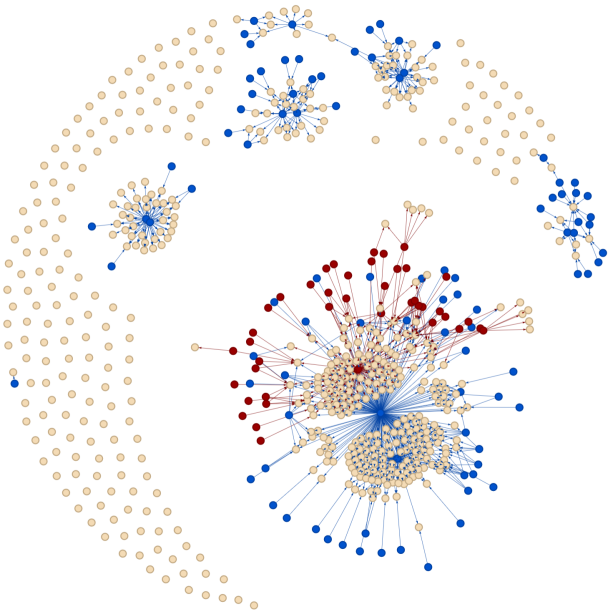


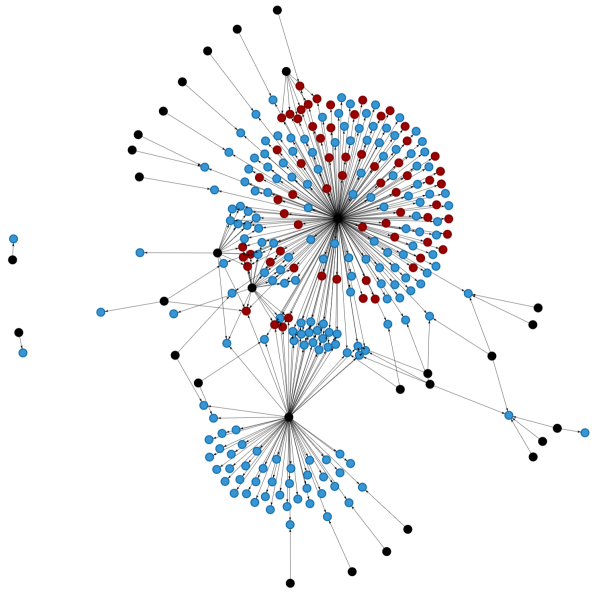
Fig. 3: Compliance Metrics

Commit Issue. The commit issue compliance metric assesses whether commits in the master branch that contain an issue

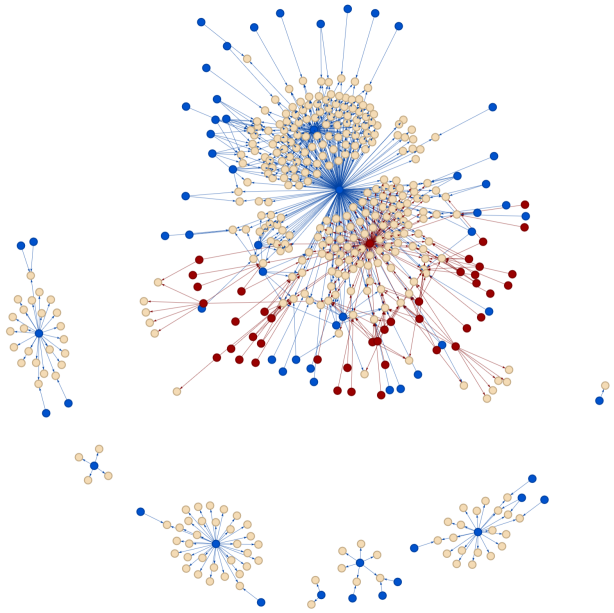
[7]Copilot issues can be found publicly at https://github.com/Copilot-Language/copilot/issues



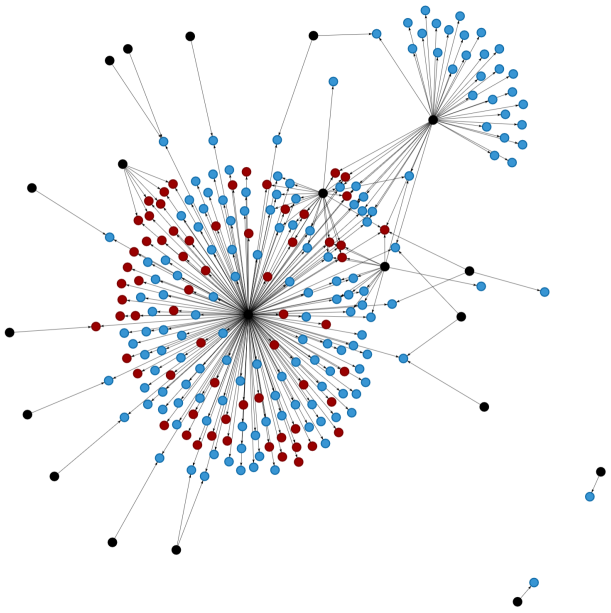
(a) File(yellow)-Issue(blue) Network



(b) Developer(black)-Issue(blue) Network



(c) File(yellow)-Issue(blue) Network Subset after May 2020



(d) Developer(black)-Issue(blue) Network after May 2020

Fig. 2: Copilot's Collaboration and Communication Networks. Developers (black), Issues (blue or red) are shown. Red issue nodes were used for issue compliance metrics, while the commit compliance metric uses all issues.

number on their message have a pair branch commit with also the same issue number (from which the commit was merged). In the previous section Figure 2c, we showed that commit messages consistently contained traceability information in their commit messages, which allowed us to connect files to issues. In figure 3 we can further see all master branch commits contained a paired branch commit referencing the same issue.

For this particular metric, full compliance was expected: this policy was introduced in the project very early on, pre-dating the Class D process, and both developers and managers became very used to following this requirement over the years. The decision to introduce it in Copilot's plan for compliance with NPR7150 emerged from the positive experience applying the requirement in multiple teams over large codebases, including with Copilot itself.

Milestone. The milestone issue compliance metric checks if the given issue is assigned a milestone. We can see from figure 3 this is indeed the case, with the exception of three issues (3 grey dots surrounding the black dot outlier in the figure). The outlier issues are #394, #395, #396. These issues were closed immediately after filing them, due to considering them invalid and duplicate (a temporary bug on GitHub itself made the issues filed not appear on the issue list, leading to some issues being filed multiple times).

Status. The status issue compliance metric measures the change request process. Specifically, it measures if status tags with different labels follow the expected order towards the issue completion. As noted before, we verified here only the "red issues" in figure 2d. We can see the majority of the issues complied to the status label. The exceptions were #332 (0.85 compliance), #278 (0.71 compliance), and #394, #395, #396 (0.14 compliance). The lowest compliance issues are consistent with the no milestone compliance, and due to the same bug on GitHub. Issues #332 and #278 reflect errors on the part of the change manager (co-author of this work) in enforcing compliance with the process. The former was caused by the issue being left in Verification state after the code associated with the issue was merged (the issue was never marked as 'Closed'). The latter non-compliance was caused by the transition labels being applied in an incorrect order when going from an issue being scheduled for a milestone to a developer actively working on the issue.

Travis. The last compliance metric verified if a given issue contained in any of its comments a reference to a Travis build. Once again, Copilot complied with the process in most issues, with the exception of issues #394, #395, and #396, which, as detailed earlier, were immediately closed after filing them.

C. Mapping Software Compliance Metrics to Source Code Files

Apart from trying to determine compliance with the project plans at the issue level, we can use Kaiaulu to investigate deeper problems that may lead to frequent process deviation. For example, we can use the same ideas and process to determine if there are files that are more susceptible to bugs, or that are frequently associated with process deviations. To do so, we map the commit and issue compliance metrics to files, and compare them against churn and bug count. Figures 4 and 5 show the correlation between the defined compliance metrics: Churn and Bug Count. The compliance metrics range from 0 to 1, where 1 indicates full compliance, as previously defined in the method section. In total, N = 358 source code files were analyzed in respect to the compliance metrics. We omitted the compliance between master and development branches, as all master commits in Copilot contained at least one branch merged onto the master branch with a matching issue number.

We can observe from Figures 4 and 5 that all files had Milestone and Travis compliance. Note, however, that in Figure 3 we noted some issues did *not* have Milestone and Travis compliance. This discrepancy in compliance is possible, because the issues which were non-compliant did not have *file changes committed to them*.

For the Status compliance metric, however, file changes were made to non-compliant issues. By inspecting Figures 4 and 5 Mean Status Compliance, we can see that, irrespective of *file changes* being made on less compliant issues, their Churn and Bug Count was similar to compliant issues. This may suggest that small deviations in compliance are tolerable in the process; that is, they do not lead to a higher (or lower) incidence of bugs or to an increase in the number of changes. An extension to this analysis would be to observe if fully non-complaint *file changes* (i.e., with compliance metric of or near zero) led to higher churn or bugs. However, since no such case occurred in Copilot after it implemented the process, we defer this analysis for future work.

Lastly, we can see that two files had substantially higher Churn in Figure 4. These two files were *copilot-theorem/src/Copilot/Theorem/What4.hs* and *copilot-theorem/src/Copilot/Theorem/What4/Translate.hs*. These files had substantially more churn than others because the modules they implemented are large and complex in nature, and definitions were moved from one file to the other at some point during development.

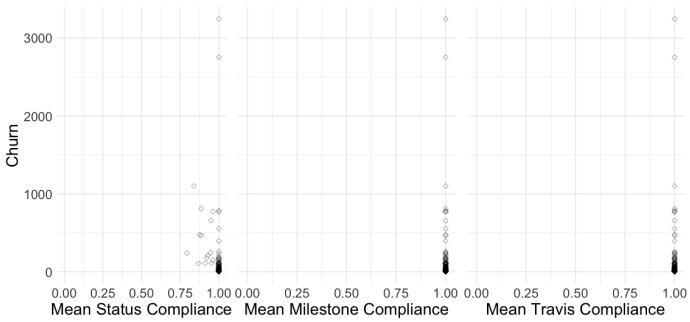


Fig. 4: File Churn vs Process Compliance Metrics

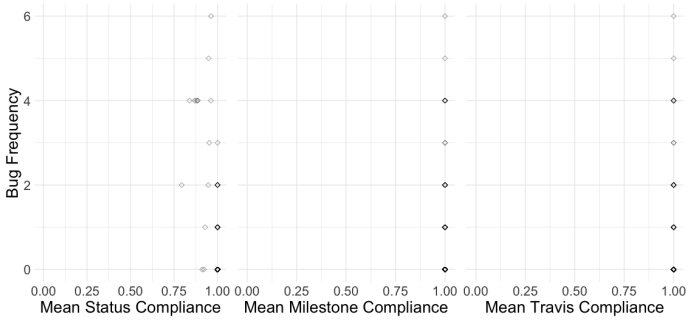


Fig. 5: File Bug Frequency vs Process Compliance Metrics

V. Conclusion and Future Work

In this work, we demonstrated how NASA's NPR 7150.2 software requirements could be instantiated as a set of process rules, which in turn could be mined to be evaluated against software compliance metrics. Because both Copilot (the use case), and Kaiaulu (the mining software repository tool on which the metrics are built), are open source, they can serve as a starting point for similar or more elaborate metrics to be used in open-source projects building on our work. In future work, we intend to refine and increase the number of software compliance metrics, extend the coverage to both NASA's NPR 7150.2 and existing open-source project practices, and evaluate other static code analysis metrics, including size (e.g. LOC), architectural and collaboration metrics.

Although the rules implemented by Copilot, and some of the metrics chosen, are specific to this project and to NASA, the ideas presented in this paper are reusable and applicable to other projects. Metrics such as the existence of a milestone for all issues addressed, referencing issues in commits, and always having a successful build report, are standard in software engineering and could find uses outside of NASA. Some of the ideas presented could easily be adapted to other open-source projects, such as searching the PR for a pre-acceptance report that ensures that certain aspects of the proposed solution have been checked. For example, Apache Camel Pull Requests[8] includes a checklist similar to Copilot's Issue Checklist (from which we derived the Travis Compliance Metric). Camel's checklist requests the developer filing the checklist to check that certain tasks were performed, like ensuring that the commit is in the correct branch, and that an issue was created for any substantial change. These items, much like was showcased using Copilot in this work, could be automated using our proposed method. Issue Templates could also be an alternative item for evaluation.[9][10] Finally, existing version control systems that support open-source development, such as GitHub, can implement some rules that prevent merging of changes. Our goal is to present a solution that can be used with any system, and to support not just the development and change acceptance efforts, but also posterior auditing.

[8]https://github.com//apache/camel/pull/10787

[9]https://docs.github.com/en/communities/using-templates-to-encourage-useful-issues-and-pull-requests/configuring-issue-templates-for-your-repository

[10]https://github.com/rstudio/rstudio/issues/new/choose

References

[1] RTCA, "Do-178c, software considerations in airborne systems and equipment certification," 2011.

[2] I. E. Commission, "Medical device software – software life cycle processes. international iec standard 62304," 05 2006.

[3] A. M. Lemos, C. C. Sabino, R. M. F. Lima, and C. A. L. Oliveira, "Using process mining in software development process management: A case study," in *2011 IEEE International Conference on Systems, Man, and Cybernetics*, pp. 1181–1186, 2011.

[4] C. Paradis and R. Kazman, "Building the msr tool kaiaulu: Design principles and experiences," in *Software Architecture* (P. Scandurra, M. Galster, R. Mirandola, and D. Weyns, eds.), (Cham), pp. 107–129, Springer International Publishing, 2022.

[5] V. Rubin, C. Günther, W. Aalst, E. Kindler, B. Dongen, and W. Schäfer, "Process mining framework for software processes," pp. 169–181, 05 2007.

[6] C. Jensen, "Data mining for software process discovery in open source software development communities," pp. 96–100, 01 2004.

[7] B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and W. M. P. van der Aalst, "The prom framework: A new era in process mining tool support," in *Applications and Theory of Petri Nets 2005* (G. Ciardo and P. Darondeau, eds.), (Berlin, Heidelberg), pp. 444–454, Springer Berlin Heidelberg, 2005.

[8] W. Poncin, A. Serebrenik, and M. Brand, "Process mining software repositories," pp. 5–14, 03 2011.

[9] S. Bala and J. Mendling, *Monitoring the Software Development Process with Process Mining*, pp. 432–442. 07 2018.

[10] P. Juneja, D. Kundra, and A. Sureka, "Anvaya: An algorithm and case-study on improving the goodness of software process models generated by mining event-log data in issue tracking systems," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 53–62, 2016.

[11] K. A. Qamar, E. Sulun, and E. Tuzun, "Towards a taxonomy of bug tracking process smells: A quantitative analysis," in *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 138–147, 2021.

[12] c. Eren, K. Şahin, and E. Tüzün, "Analyzing bug life cycles to derive practical insights," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, (New York, NY, USA), p. 162–171, Association for Computing Machinery, 2023.

[13] M. Gupta and A. Sureka, "Nirikshan: Mining bug report history for discovering process maps, inefficiencies and inconsistencies," in *Proceedings of the 7th India Software Engineering Conference*, ISEC '14, (New York, NY, USA), Association for Computing Machinery, 2014.

[14] T. Lehtonen, V.-P. Eloranta, M. Leppänen, and E. Isohanni, "Visualizations as a basis for agile software process improvement," in *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, vol. 1, pp. 495–502, 2013.

[15] A.-L. Mattila, T. Lehtonen, H. Terho, T. Mikkonen, and K. Systä, "Mashing up software issue management, development, and usage data," in *2015 IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering*, pp. 26–29, 2015.

[16] J. S. Saltz, "Visualizing kanban work: Towards an individual contributor view.," in *AMCIS*, 2019.

[17] A. Berti, S. J. van Zelst, and W. van der Aalst, "Process mining for python (pm4py): Bridging the gap between process- and data science," 2019.

[18] H. Mumtaz, C. Paradis, F. Palomba, D. A. Tamburri, R. Kazman, and K. Blincoe, "A preliminary study on the assignment of github issues to issue commenters and the relationship with social smells," in *Proceedings of the 15th International Conference on Cooperative and Human Aspects of Software Engineering*, CHASE '22, (New York, NY, USA), p. 61–65, Association for Computing Machinery, 2022.

[19] V. Driessen, "A successful git branching model." https://nvie.com/posts/a-successful-git-branching-model/, 2010.