SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

What Can We Learn from Quality Assurance Badges in Open-Source Software?

Feng LI[1], Yiling Lou[2], Xin TAN[3], Zhenpeng CHEN[4], Jinhao DONG[1], Yang LI[1], Xuanzhi WANG[1], Dan HAO[1*] & Lu ZHANG[1]

[1]*MoE Key Lab of HCST, School of Computer Science, Peking University, Beijing* 100871*, China;*
[2]*Department of Computer Science, Purdue University, West Lafayette, IN* 47907*, USA;*
[3]*School of Computer Science & Engineering, Beihang University (BUAA), Beijing* 100191*, China;*
[4]*Department of Computer Science, University College London (UCL), London* WC1E 6BT*, UK*

Abstract In the development of open-source software (OSS), many developers use badges to give an overview of the software and share some key features/metrics conveniently. Among various badges, quality assurance (QA) badges make up a large proportion and are the most prevalent because QA is of vital importance in software development, and ineffective QA may lead to anomalies or defects. In this paper, we focus on QA badges in open-source projects, which present quality assurance information directly and instantly, and aim to produce some interesting findings and provide practical implications. We collect and analyze 100,000 projects written in popular programming languages from GitHub and conduct a comprehensive empirical study both inside and outside QA badges. Inside QA badges, we build a category for all QA badges based on the properties they focus on, which shows the types of QA badges developers use. Then, we analyze the frequency of the properties that QA badges focus on, and property combinations, too, which present their use status. We find that QA badges focus on various properties while developers give different preferences to different properties. The use status also differs between different programming languages. For example, projects written in *C* focus on security to a great extent. Our findings also provide implications for developers and badge providers. Outside QA badges, we conduct a correlation analysis between QA badges and some software metrics that have potential relationships with code quality, contribution quality, and popularity. We find that QA badges have statistically significant correlations with various software metrics.

Keywords quality assurance, badge, open-source software, code quality, empirical study

Citation Li F, Lou Y, Tan X, et al. What Can We Learn from Quality Assurance Badges in Open-Source Software?. Sci China Inf Sci, for review

1 Introduction

Open-source software (OSS) is prevalent [1–3], providing a range of services and products for companies, governments, and educational organizations [3]. Meanwhile, various OSS development platforms have been developing rapidly, such as GitHub[1)] To help developers better maintain and understand software evolution, OSS development platforms often encourage the documentation of each project with a basic introduction. For example, on the most popular OSS platform, GitHub, developers often maintain a profile README to describe an overview of each project, such as its basic information, installation guidelines, or announcements, which can help users and contributors get to know the project more easily.

In addition to the traditional textual descriptions, more and more projects have included *badges* in their README files. A *badge* is an image like build passing, describing a certain property/properties of the project. Each badge is supported by an external service provider, who provides the image and presents information on it. Typically, a badge can describe information on QA, dependency management, etc.

The increasing popularity of badges has also gained attention from researchers. In particular, Trockman et al. [4] present the first large-scale empirical study on badges in the npm ecosystem and confirm

* Corresponding author (email: haodan@pku.edu.cn)
1) https://github.com/. Since this paper covers many tools and websites, we do not give their links whenever they occur but put them in Appendix A together.

that many badges are indeed reasonably reliable signals for both users and contributors. Their work further summarizes five common types of badges, detailing QA, dependency management, information, popularity, and support. Among them, QA badges are the most prevalent and make up the largest proportion of all badges. For example, 31.5% of their studied projects adopt the *build status* badges that are provided by Travis CI. Such an observation is expected since QA is essential for OSS. To date, quality assurance has been studied extensively in the literature [5–9], and various QA tools have been proposed and integrated into OSS platforms, which further leads to the prevalence of QA badges.

However, although QA badges have been widely used in practice, their use status in the wild or their practical impact on quality assurance remains unknown. While Trockman et al. [4] performed the first study on overall badges, no previous work has focused specifically on QA badges. Inspired by them, in this paper, we present the first comprehensive study on QA badges in OSS to investigate their use status and implications in practice. We first construct a large-scale dataset consisting of 57,363 QA badges collected from the top 100,000 popular GitHub projects in 10 popular programming languages. Then, we build a categorization of the properties that QA badges focus on. Based on these data, we conduct our empirical study both inside and outside QA badges.

First, we conduct analyses **inside** QA badges. Specifically, facing various QA badges in an open-source ecosystem, we are curious about what software properties (e.g., maintainability, security) developers are focusing on and the use status of different badges and properties. Since we do not even know what kinds of QA badges developers commonly use, we first describe the full picture of existing QA badges by building a category of the properties that QA badges focus on, which can help in understanding well the current state of QA in OSS projects. Then, after having an overall understanding of QA badges and building a category, we observe the frequency of each badge category. We aim to find their use patterns, including the use status of properties, property combinations, and use status in projects written in different programming languages. We find that QA badges focus on various properties while developers assign different preferences to different properties. The use status in different programming languages also differs; for example, projects written in *C* focus on security to a large extent. Our findings also have implications for developers and badge providers.

Second, besides analyses inside QA badges, we conduct analyses **outside** QA badges, that is, to investigate the correlation between QA badges and some software metrics. We select some metrics that should have a potential relationship with software code quality, contribution quality, and popularity. We expect the metrics to have correlations with QA badges or be impacted by QA badges. We conduct correlation analysis on QA badges and software metrics, and the experiments are conducted (1) between projects with and without QA badges and (2) between projects with QA badges and their previous versions, where QA badges have not been introduced. We find that QA badges are correlated with bug-related metrics, metrics related to testing code contributions, and popularity-related metrics.

In summary, we contribute (1) a large dataset containing 100,000 popular GitHub projects written in 10 popular programming languages and their 57,363 badges, (2) a categorization of all the properties that QA badges focus on, (3) a large-scale and in-depth analysis of the use status of QA badges/properties, and (4) analyses of the correlation between QA badges and other project metrics that have potential relationships with software code quality, contribution quality, and popularity. Our replication package is available on our website **https://github.com/Spiridempt/Badge**.

2 Background, Methodology, and Research Questions

In this section, we briefly introduce badges and QA badges and then present our research methodology and research questions.

2.1 Background

Badge. In OSS development practice, to give an overview of the project and share some overall information, developers usually create a profile README. As the README is often the first item a visitor will see when visiting the repository, it is important to write it well. README files typically address (1) what the project does, (2) why the project is useful, (3) how users get started with the project, (4) where users can get help with the project, and (5) who maintains and contributes to the project.

To show such information clearly and conveniently, lots of projects introduce badges in their README files. Specifically, a badge is a special image like
, and each badge usually has a service provider

that provides the specific information of a project, and the service provider also provides the image of a badge. For dynamic badges, the information is dynamically updated by the provider and the image is also dynamically updated. Then, through the image link in the README file, the project can present current information. Typical examples of dynamic badges are those showing the number of downloads or the current code coverage. For static badges, the information is often fixed, and the image need not be updated dynamically. Typical examples of static badges are those badges showing the license or giving links for donations.

QA badge. Among the many badges, some are related to the QA of the project. In this paper, we define these kinds of badges as QA badges. For example, build passing is a badge provided by Travis CI, which shows the current build status of the project, and coverage 97% is a badge provided by Coveralls, which shows the current code coverage.

The reasons developers use QA badges include but are not limited to: (1) Using badges is a better way to present characteristics than providing some links for inspection. (2) The metrics on badges can encourage developers to optimize their code to show better metrics (values). (3) Clear badges and good metrics can attract more contributors to participate in project development [4, 10].

In contrast to the wide use of badges in practice, few studies have investigated badges (especially QA badges) and their use in practice. To our knowledge, Trockman et al. [4] present the only large-scale empirical study on badges in the *npm* ecosystem and confirm many badges are indeed reasonably reliable signals for users and contributors. Among the various badges, QA badges are the most prevalent, making up the largest proportion of all badges because quality assurance is of vital importance in OSS. Therefore, we focus on QA badges in this paper and aim to make in-depth analyses.

2.2 Methodology and Research Questions

Before analyzing QA badges in open-source projects, we illustrate our experimental methodology and research questions in this section. Figure 1 shows the overview. Note that all data and analyzing scripts are on our website for reproduction.

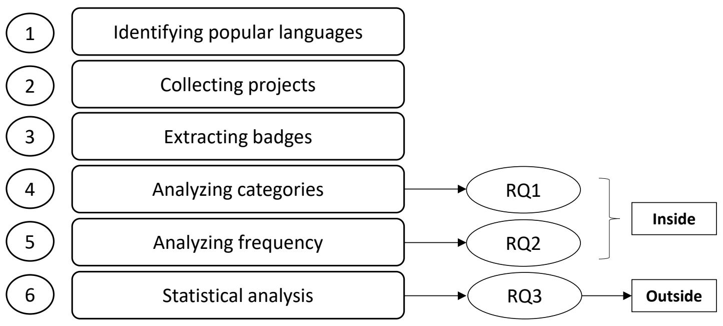

Figure 1 Methodology Overview.

Following previous study [4], we collect a dataset from GitHub. Since GitHub is widely used by developers to host open-source projects, it reflects real-world development practice.

Step 1: Identifying popular languages. As different programming languages have different QA characteristics, it is natural to consider programming languages when collecting data (i.e., stratified sampling [11]). According to the State of the Octoverse Report[2)], we select the top 10 most popular programming languages in 2021, i.e., *JavaScript*, *Python*, *Java*, *TypeScript*, *C#*, *PHP*, *C++*, *C*, *Shell*, and *Ruby*.

Step 2: Collecting projects. To conduct stratified sampling, for each programming language, we select 10,000 projects with the largest number of stars, which results in amassing 100,000 projects in total. The reason we choose popular projects is that they are relatively active and mature and are expected to have good QA status, but this is a source of potential bias that may affect the degree to which we can generalize from the findings. The discussion of this threat is in Section 6. It is worth noting that in the collecting process, to avoid unclonable and repeated repositories, we skip private repositories, forked repositories, and non-software repositories as previous work [4, 12, 13] has done and stop until we succeed in gathering

2) https://octoverse.github.com/

10,000 projects for each programming language. For each of these projects, we clone it locally and collect its metadata, such as #commits, #watches, #stars, and #forks. The collecting date is Aug 5, 2021.

Step 3: Extracting badges. After manual checking, we follow previous work [4] to extract QA badges in Steps 3 and 4. Most projects use Markdown expressions in README to insert badges, i.e., [![Badge Name](Badge Image Link)](Service Link). However, some projects use HTML tags or other methods to insert badges. Therefore, to avoid the loss of badges, we first convert all README files to an HTML format using Pandoc[3]. With this step, all kinds of badges are said to be converted to *img* tags in HTML files.

Then, we regard "Badge Name + Domain Name (in Service Link)" as the identifier of a badge. Note that images other than badges also have three corresponding components (Image Name, Image Link, and Image Service Link). Therefore, this procedure introduces lots of other images, which cannot be distinguished automatically. The manual process is stated in Step 4.

We need to distinguish QA badges and non-QA badges further. If the information that a badge focuses on contains any aspects related to QA (e.g., code coverage, defects, code smells), it is regarded as a QA badge. Otherwise, if the information contains nothing about QA (e.g., version, dependency, chat room), it is regarded as a non-QA badge. The manual process is also stated in Step 4.

Next, we introduce our research questions (RQ1–3) and their corresponding steps (Steps 4–6).

First, we conduct analyses **inside** QA badges. Specifically, facing various QA badges in an open-source ecosystem, we are curious about what aspects (properties) they focus on and their use status. This leads to RQ1 and RQ2.

RQ1: Category. *What kinds of QA badges do developers use?* Since we do not know what kinds of QA badges developers commonly use, RQ1 first describes a full picture of existing QA badges. Specifically, we build a category of aspects (properties) that QA badges focus on, which can help in understanding the current state of quality assurance in open-source projects well.

Step 4: Constructing the category. As illustrated in Step 3, using "Badge Name + Domain Name (in Service Link)" as the identifier of a single badge, we acquire in total 24,982 "badges" (actually images) from all projects. Intuitively, images that are not badges were hard to repeat many times. Therefore, from the total images, we first choose 500 randomly that occur less than 10 times and find that 93.8% (469/500) of them are not QA badges. At the same time, 31 QA badges can be merged into other badges that occur at least 10 times. Then, to make the results more representative while reducing our manual efforts, two of the authors review the remaining 1,033 badges (images) that occur at least 10 times and merge and classify them.

We aim to build a category of existing QA badges and analyze what kind of QA badges developers use and what aspects developers focus on. Therefore, we follow an open coding procedure [14]: (1) Two of the authors read and reread all 1,033 identified badges (images), check the links behind each image, and exclude 63 non-badges and 682 non-QA badges. (2) Due to different naming rules, many badges are, in fact, the same. For example, developers may give Travis CI build status badges different names, such as "Travis Build" and "Build Result"; therefore, during the process, we merge them manually. Detailed explanations are given in Section 3. (3) For subsequent analysis, we build a category based on the **properties** that QA badges focus on, such as code coverage and code smells. We first jointly build the category based on 30% of the QA badges. We generate initial nodes by checking the focused properties of each badge. (4) We group the initial nodes into inner nodes that are conceptually similar. For example, Code Coverage and Mutation Score are both related to Testing Code. (5) Then, we keep modifying the nodes and merging similar nodes. (6) Finally, we define the final category. After defining the final category, the two authors classify each of the remaining 70% of QA badges into one or more nodes. The inter-rater agreement was 0.891 (Cohen's kappa), indicating almost perfect agreement [15] and demonstrating the reliability of our procedure. All encountered conflicts were resolved through discussions. In particular, as many badges with different names can be merged into one badge, during the process, we also design a text-matching approach to automatically merge those badges.

Two things are worth noting. First, after our procedure, most badges can be merged, and the total number of QA badges is 45, which is consistent with previous work [4]. Second, the category is based on **properties** that badges focus on. A detailed explanation is given in Section 3.

RQ2: Frequency. *What is the use status of QA badges?*

Step 5: Analyzing frequency. After having an overall understanding of QA badges and building a

[3] https://github.com/jgm/pandoc

category, it is natural to observe the frequency of each badge category. We aim to find their use patterns, including the use status of properties, property combinations, and the use status in projects in different programming languages. This research question helps us have an in-depth understanding of the use status of QA badges.

Besides analyses inside QA badges, we also conduct analyses **outside** QA badges to investigate the correlation between QA badges and some software metrics. This procedure is as outlined in RQ3.

RQ3: Correlation. *Are QA badges correlated with some software metrics?*

In RQ3, we select some metrics that should have a potential relationship with software code quality, contribution quality, and popularity, which we expect to have correlations with QA badges. We conduct a correlation analysis between QA badges and these software metrics.

Step 6: Statistical analysis.

Considering the reasons why developers use QA badges, QA badges may have correlations with code-quality-related, contribution-quality-related, and popularity-related characteristics. Therefore, we first choose the following metrics that have a potential relationship with code/contribution quality and popularity and show how to collect them.

Number of bug-fixing commits. Following previous work [16, 17], we use “fix” and “bug” as keywords (the case is insensitive) to search all historical commit messages and use the number of matched commits as its indicator.

Number of bug issues. QA badges may help developers improve code quality, resulting in fewer bug issues. On the other hand, QA badges may accelerate the exposure of bugs and bug issues. Following previous work [16], if the label of an issue contains keyword “bug” (the case is insensitive), we regard it as a bug issue.

Lines of Testing Code (TLOC) [4]. We use CLOC[4)] to calculate the total and for testing lines of code. Specifically, due to the complex code structures in different projects, we simply regard file paths that contain keyword “test” (case-insensitive) as testing files. We calculated only code written in the primary language.

Number of commits containing testing code [4] reflects the frequency of developers contributing to improving testing code, which indicates contribution quality. The intuition is that QA badges set expectations of contribution quality for new contributors. As previous work [4] says, “Pull requests with new functionality tend to include new tests, so as not to decrease coverage.” We also regard file paths containing keyword “test” (case-insensitive) as testing files.

Numbers of Stars, Forks, and Watchers [4, 16, 18] reflect the software popularity. They can be directly obtained through the project homepage.

The above metrics are commonly used in previous work and have a potential relationship with code quality, contribution quality, popularity, etc.

Because many other key features may have correlations with our metrics, following previous studies [16, 19, 20], we include the following variables as control variables: Lines of code (LOC), Number of commits, Number of issues, Number of contributors, and Age. In our statistical analysis process, we regard them as covariates to obtain reliable results.

The analysis consists of two parts. In the first part, we analyze projects with/without QA badges. We adopt Generalized Additive Models (GAM) to perform correlation analysis. In the second part, we conduct a longitudinal analysis between projects having QA badges and their previous versions, in which the QA badges have not been introduced, and GAMs are performed. The detailed statistical procedures are illustrated in Section 5.

3 RQ1: Categories of the aspects that QA badges focus on

In RQ1, we conduct analyses **inside** QA badges, i.e., to see what kinds of QA badges developers use in open-source communities.

3.1 Categorization

After manually checking all badges that occur at least 10 times, we obtain 29,845 projects with at least one QA badges among all the 100,000 projects, indicating an unignorable percentage (i.e., 29.8%) of

4) https://github.com/AlDanial/cloc

projects with QA badges.

For each of the identified 45 QA badges, we check its functionality and service provider to learn more details. We find that QA badges usually focus on certain code quality property/properties. Here, code quality refers to any aspects related to software defects, maintainability, testing effectiveness, efficiency, clarity, etc. For example, a badge shows the current code coverage like coverage 80%. It is provided by Coveralls, a web service to help developers track the code coverage over time. Another badge shows the number of vulnerabilities currently like vulnerabilities 0. It is provided by Snyk, a platform that focuses on software security.

To dive into various QA badges, we define code quality properties as the various aspects that QA badges focus on and organize them into different categories. Because different tools focus on different code quality properties, they have overlaps to some extent. Also, many tools provide an "overall" badge, which is an overall evaluation of multiple properties. We manually check each QA badge following the procedures in Step 4 in Section 2.2 and give a category of all kinds of properties that QA badges care about in Figure 2. Note that the number after each node means the number of badges it is covered by.

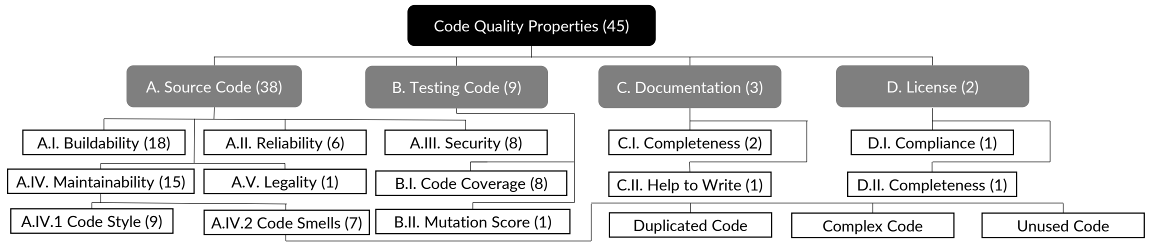

Figure 2 Category of code quality properties that QA badges focus on.

The root category is divided into four categories, Source Code, Testing Code, Documentation, and License, reflecting the component that the property of a badge is related to.

Source Code (A) contains properties related to the source code in a project, including *Buildability (A.I)*, *Reliability (A.II)*, *Security (A.III)*, *Maintainability (A.IV)*, and *Legality (A.V)*.

Buildability (A.I) represents badges that show the current build status, which reflects whether the current code under test is correct. Nowadays, many projects adopt continuous integration/continuous delivery (usually abbreviated as CI/CD) to automate the building, testing, and deployment stages of software development. These CI/CD tools often integrate with GitHub and when specific events happen (e.g., commit, pull request), they start a new building process and return the build status. To let developers know their build status more clearly, these tools often provide a badge that show the latest building result. For example, Travis CI provides a badge like build passing. This kind of badges gives quality assurance by instant feedback of build results, which could let developers fix their code soon if the build fails.

Reliability (A.II) represents badges that show the software defects detection results, which could cause a program to crash or produce invalid output. A typical example is the badge provided by DeepScan. One of its evaluation metrics is software bugs.

Security (A.III) represents badges that check the potential security issues (vulnerabilities) in a project. Vulnerability is a term in computer security, which is a weakness that can be exploited by a threat actor, such as an attacker. A typical example is the badge provided by Snyk, which shows the number of vulnerabilities.

Maintainability (A.IV) represents badges that focus on evolvability, modifiability, technical debt, and code smells [21][5]. Specifically, it has two main sub-categories in the context of existing QA badges, *Code Style (A.IV.1)* and *Code Smells (A.IV.2)*. *Code Style (A.IV.1)* represents better code formats. Many tools help developers reorganize their code to comply to specific code style, e.g., Prettier, StyleCI. *Code Smells (A.IV.2)* refers to any symptom in the source code of a program that possibly indicates a deeper problem (e.g., duplicated code, over-complex code, unused code). Platforms such as Code Climate and Codacy provide such badges.

Legality (A.V) contains problems that are related to intellectual properties or violate GDPR[6]. Plat-

[5] https://en.wikipedia.org/wiki/Maintainability
[6] GDPR refers to the General Data Protection Regulation (EU) 2016/679

forms such as SymfonyInsight provide badges focusing on Legality.

Testing Code (B) consists of *Code Coverage (B.I)* and *Mutation Score (B.II)*, both of which measure the testing adequacy.

Code Coverage (B.I) is a testing adequacy measurement that calculates what fraction of code is covered by a test suite. Tools like Coveralls provide badges to present the current code coverage to help developers optimize their test suite.

Mutation Score (B.II) is another testing adequacy measurement, which calculates the percentage of program mutants killed/revealed by a test suite. For example, Stryker Mutator gives a badge on mutation score.

Besides source code and testing code, documentation and license are also important components in open-source software.

Documentation (C) refers to written text or diagram accompanying software or inserted in the source code, which either describes how to use the software or how the software works. Many tools focus on the software documentation instead of the code. Inch CI focuses on *Completeness (C.I)* and gives a badge showing the overall evaluation on documentation. Read the Docs *Help to Write (C.II)* documentation and gives a badge showing the current status of documentation generation.

License (D) allows open-source software to be freely used, modified, and shared. The license checking mainly consists of two categories, *Compliance (D.I)* and *Completeness (D.II)*. Typical examples are FOSSA for the former and CII Best Practice for the latter.

It is worth noting that the category in Figure 2 is based on properties rather than badges. In other words, any QA badge can be broken into one or more nodes in Figure 2. The detailed classification results of each badge are on our website.

Finding: Quality assurance badges focus on various aspects in open-source software, including buildability, reliability, security, maintainability, testing adequacy.

Implications: (1) For developers. Software developers could learn that there exist various different QA badges that focus on different aspects of software. This helps developers notice the common practice of the usage of QA badges. In fact, the intention of adopting badges includes presenting project characteristics, motivating developers to improve the shown metrics, recruiting more contributors, etc. Therefore, when developing open-source software, developers could also choose badges that cover as more aspects as possible. (2) For badge providers. Because our category is merely based on existing QA badges, we could notice the gap between the category and the underlying complete taxonomy. As we can see, many aspects have not been covered by existing QA badges. For example, dependency bugs, concurrency bugs, performance bugs, efficiency bugs, etc. are frequent bug types [22–24] and perhaps need tool support. Badges related to reliability and security are small in number and need to be improved. Badges showing more test adequacy criteria (e.g., MC/DC [25], TCQA [26]) are also necessary.

3.2 Discussion

Discussion on badge providers. During our analysis process, we find that many platforms provide more than one kind of badges for developers. To make an in-depth analysis, we discuss this scenario here. On one hand, instead of only providing an "overall" badge that makes a comprehensive evaluation, many tools give separate badges that can present detailed information for different properties. For example, Code Climate provides badges for code smell, code coverage, and overall evaluation, respectively. SonarCloud provides badges for bugs, code smells, and code coverage, respectively. On the other hand, some CI/CD platforms not only build the project automatically and give a Build Status badge, but also conduct some analysis and provide some other QA badges. For example, Scrutinizer provides a badge to show real-time building status and two badges for the analysis results of code coverage and code smells.

Discussion on programming languages. Furthermore, we present the statistics of QA badges with various programming languages in Table 1. In Table 1, the first row shows the programming languages, and the remaining four rows show the number and ratio of projects containing QA badges and badges, respectively.

From Table 1, on average, about 40.3% open-source projects adopt badges, while 29.3% projects adopt at least one QA badge among the total 100,000 projects, which indicates developers pay attention to

Table 1 Usage status of quality-assurance badges in different programming languages.

Programming Language	*JavaScript*	*Python*	*Java*	*TypeScript*	*C#*	*PHP*	*C++*	*C*	*Shell*	*Ruby*	Total
#Projects with QA badges	3,944 (39.4%)	3,363 (33.6%)	2,347 (23.5%)	4,116 (41.2%)	2,108 (21.1%)	3,852 (38.5%)	2,491 (24.9%)	1,793 (17.9%)	1,281 (12.8%)	4,023 (40.2%)	29,318 (29.3%)
#Projects with badges	5,454 (54.5%)	4,338 (43.4%)	4,429 (44.3%)	5,563 (55.6%)	3,419 (34.2%)	4,853 (48.5%)	3,238 (32.4%)	2,320 (23.2%)	2,070 (20.7%)	4,571 (45.7%)	40,255 (40.3%)

quality assurance. This is also consistent with previous study [4] that QA badges occupy a large part among all badges. Further, different languages have different usage status. In *JavaScript*, *TypeScript*, *PHP*, and *Ruby*, the number of projects that use QA badges is larger than that in other languages. In particular, among *TypeScript* projects, about 41% adopt QA badges. However, in *C* and *Shell*, fewer projects take the use of QA badges. On one hand, this reflects that differences exist in the development practice among different languages. One the other hand, this may also reflect the demand of satisfying QA badges in some languages.

Finding: On average, in our dataset, 29.3% projects adopt at least one quality-assurance badge. At the same time, different programming languages have different adoption ratio, ranging from 12.8% to 41.2%.

4 RQ2: Usage frequency of QA badges

In RQ2, we analyze the use status of QA badges. In particular, we only consider projects using at least one QA badge from now on.

4.1 Overall analysis

Table 2 Number of projects that focus on different aspects.

Badge Type	A. Source Code	B. Testing Code	C. Documentation	D. License	Total
#Projects with badges	28,673	9,687	1,464	375	29,318

Table 3 Number of badges in each project.

Badge Type	A. Source Code				B. Testing Code				C. Documentation				D. License				Total			
Metrics	min	max	avg	total	min	max	avg	total	min	max	avg	total	min	max	avg	total	min	max	avg	total
#Badges	0	125	1.47	43,063	0	37	0.42	12,199	0	9	0.05	1,564	0	3	0.02	537	1	125	1.96	57,363

Tables 2 and 3 present the overall use status of four main categories of QA badges. In Table 2, the first row shows the categories and the second row shows the number of projects having the corresponding QA badges. From the table, among the projects that have at least one QA badge, 97.80% (28,673/29,318) use QA badges that focus on source code, while 33.04% (9,687/29,318) use QA badges that focus on testing code. The usage of QA badges focusing on documentation and license is relatively less frequent. One one hand, this is consistent with common sense that source code should be paid more attention. On the other hand, badge providers also tend to design more QA badges related to source code and testing code.

Table 3 presents the statistics of the number of QA badges focusing on different categories within a project. For A. Source Code, the number of QA badges ranges from 0 to 125, with an average 1.47. After carefully checking, there exists a project called *elasticsearch-sql* whose README file contains the Build Status badges of all its 125 versions. In total, the number of QA badges ranges from 1 to 125, with an average 1.96. Based on the table, we can find that each project often uses more than one QA badge, and QA badges related to Source Code are used the most extensively.

Finding: In our dataset, the usage of badges related to source code and testing code is the most frequent. One project often uses more than one QA badge.

Then, we are curious about the usage patterns of various QA badges. Specifically, for the categories in Section 3, we analyze their frequency and investigate what aspects developers focus on.

Because A. Source Code and B. Testing Code have many sub-categories, we analyze the categories in the following granularities: Buildability (A.I), Reliability (A.II), Security (A.III), Code Style (A.IV.1), Code Smells (A.IV.2), Legality (A.V), Code Coverage (B.I), Mutation Score (B.II), Documentation (C), and License (D).

Table 4 #Occurrences of code quality properties (categories).

Property name	Buildability (A.I)	Code Coverage (B.I)	Code Smells (A.IV.2)	Security (A.III)	Code Style (A.IV.1)
#Occurrences and Ratio	27,486 (93.75%)	9,680 (33.02%)	4,178 (14.25%)	2,354 (8.03%)	2,245 (7.66%)
Property name	Reliability (A.II)	Documentation (C)	License (D)	Legality (A.V)	Mutation Score (B.II)
#Occurrences and Ratio	1,784 (6.08%)	1,464 (4.99%)	375 (1.28%)	277 (0.94%)	19 (0.06%)

The results are presented in Table 4, which shows the property (category) name, how many projects it occurs in, and the ratio of occurrence, respectively. The code quality properties are listed based on the descendent order of their occurrence. As we can see, the most frequent property is Buildability, which represents badges that show the current build status. In fact, it is common practice for open-source developers to adopt continuous integration platforms. At the same time, mainstream platforms usually design badges for developers to check real-time build status conveniently. Therefore, it becomes the most frequent QA badge.

Code Coverage, as an effective measurement of testing adequacy, also receives much attention among all properties. Besides the importance of code coverage in practice, another possible reason is that the calculation of code coverage is relatively simple and easy to be integrated with a CI platform, so, many analysis tools take code coverage as their basic functionality.

Developers mainly focus on Source Code-related problem, which is consistent with our intuition. Among them, Code Smells, Vulnerabilities (Security), and Code Style are the most popular ones. Avoiding code smells and using good code style are both approaches to achieve high software quality and productivity. Many tools focus on them from various aspects. For example, badges provided by Codacy, SonarCloud, and CodeBeat use code smells as part of their analyses. Badges provided by StyleCI, Prettier, and HoundCI focus on code styling. The results also show that open-source projects attach importance to software security.

Although code coverage and mutation score are both testing adequacy measurements, the latter is rarely measured, especially compared with the former, which may indicate more efforts on mutation testing (e.g., research on addressing its limitations or tool supports) are needed.

Finding: Buildability (build status badge) is most concerned among all properties (categories), occurring in 93.75% projects. Besides, developers care about code coverage (33.02%) and source code-related properties, i.e., Code Smells (14.25%), Security (8.03%), and Code Style (7.66%). On one hand, this reflects the preference of different properties in open-source software development. On the other hand, this shows different tool support situations of badges focusing on different properties, which leads to different numbers of badges in various categories.

Implications: For developers, we provide current usage patterns of badges (properties). If developers want to follow common practice in open-source ecosystem, they could refer to our analyses. Otherwise, developers may pay attention to those less concerned properties, which could be drawbacks in their software and have large spaces to improve. For badge providers, on one hand, they could pay attention to frequently used badges (properties) and provide better services. On the other hand, they could focus on less concerned badges (properties), design possible badges, and occupy the underlying markets.

4.2 Analysis on property combinations

We further observe the combinations of properties (categories) in open-source projects. Specifically, as the combinations of two properties are more frequent than the combinations of more than two properties, and also in order to reduce the problem complexity, we consider the combinations of **two** properties in this section. Note that one property can be combined with itself if it occurs more than once in a project.

Table 5 shows property combinations and the number (and ratio) of projects they occur.

In many projects, rather than focusing on a certain property, developers often care about property combinations. It is not surprising that the most frequent combinations consist of those properties that

Table 5 #Occurrences of code quality property combinations.

Combinations	Code Coverage	Buildability	Code Coverage	Code Smells	Code Coverage	Reliability	Documentation	Reliability	Buildability	Code Style
	Buildability	Buildability	Code Smells	Buildability	Code Coverage	Buildability	Buildability	Code Coverage	Security	Code Smells
#Occurrences	7,760 (26.47%)	3,922 (13.38%)	3,645 (12.43%)	2,726 (9.30%)	2,026 (6.91%)	1,419 (4.84%)	1,239 (4.23%)	1,034 (3.53%)	1,013 (3.46%)	992 (3.38%)
Combinations	Security	Code Style	Reliability	Security	Buildability	Buildability	Security	Security	Documentation	Reliability
	Reliability	Buildability	Code Smells	Buildability	Code Coverage	Code Smells	Code Smells	Code Coverage	Code Coverage	Security
#Occurrences	992 (3.38%)	953 (3.25%)	935 (3.19%)	903 (3.08%)	878 (2.99%)	871 (2.97%)	857 (2.92%)	851 (2.90%)	799 (2.73%)	773 (2.64%)
Combinations	Code Coverage	Code Smells	Reliability	Code Coverage	Buildability	Security	Code Style	Code Smells	Security	Code Smells
	Security	Code Smells	Code Style	Code Style	Code Style	Code Style	Code Coverage	Security	Security	Reliability
#Occurrences	764 (2.61%)	722 (2.46%)	706 (2.41%)	699 (2.38%)	697 (2.38%)	695 (2.37%)	663 (2.26%)	503 (1.72%)	469 (1.60%)	366 (1.25%)
Combinations	Code Smells	Documentation	Code Coverage	Code Style	Documentation	Reliability	Code Style	Legality	Buildability	Documentation
	Code Coverage	Code Smells	Reliability	Code Style	Security	Reliability	Security	Security	License	Reliability
#Occurrences	319 (1.09%)	309 (1.05%)	291 (0.99%)	272 (0.93%)	242 (0.83%)	233 (0.79%)	216 (0.74%)	214 (0.73%)	212 (0.72%)	202 (0.69%)
Combinations	Legality	Legality	Legality	Legality	Code Style	License	Code Coverage	Documentation	Documentation	Reliability
	Reliability	Code Smells	Code Coverage	Buildability	Reliability	License	License	License	Code Style	License
#Occurrences	193 (0.66%)	183 (0.62%)	165 (0.56%)	162 (0.55%)	161 (0.55%)	158 (0.54%)	128 (0.44%)	127 (0.43%)	111 (0.38%)	107 (0.36%)
Combinations	Code Smells	License	Reliability	Buildability	Documentation	Security	Code Style	Security	Code Smells	Code Style
	Legality	Security	Legality	Legality	Documentation	License	Documentation	Legality	License	License
#Occurrences	94 (0.32%)	92 (0.31%)	84 (0.29%)	82 (0.28%)	81 (0.28%)	77 (0.26%)	67 (0.23%)	63 (0.21%)	61 (0.21%)	58 (0.20%)
Combinations	License	License	Buildability	License	Legality	License	Code Coverage	Code Coverage	Code Coverage	License
	Buildability	Reliability	Reliability	Code Coverage	Code Style	Code Smells	Legality	Documentation	Mutation Score	Code Style
#Occurrences	57 (0.19%)	52 (0.18%)	49 (0.17%)	44 (0.15%)	36 (0.12%)	20 (0.07%)	20 (0.07%)	16 (0.05%)	12 (0.04%)	11 (0.04%)
Combinations	Buildability	Reliability	Legality	Code Smells	Code Smells	Mutation Score	Legality	Security	License	Legality
	Mutation Score	Documentation	Legality	Documentation	Mutation Score	Mutation Score	License	Documentation	Legality	Documentation
#Occurrences	10 (0.03%)	5 (0.02%)	5 (0.02%)	4 (0.01%)	4 (0.01%)	2 (0.01%)	1 (<0.01%)	1 (<0.01%)	1 (<0.01%)	1 (<0.01%)
Combinations	Documentation	Code Style								
	Legality	Legality								
#Occurrences	1 (<0.01%)	1 (<0.01%)								

are frequent in Table 4. Concretely, the most frequent combination consists of Buildability and Code Coverage. In fact, many code coverage tools have been integrated with many (one or more) continuous integration (CI) platform. On one hand, this could make the adoption of code coverage tools more convenient, which means developers need not to configure their repositories to communicate to those tools separately. On the other hand, the combination enables instant feedback once new code is committed to the repository and a new build process starts on the CI platform. The integration of CI and other tools also leads to the high frequency of the combination between Buildability and other properties in Table 5.

It may be surprising that so many projects use multiple build status badges which focus on Buildability, including badges from the same platforms or different platforms.

First, one project may have different build types, which differ in operating systems (e.g., Linux, Windows, macOS), versions (e.g., stable, nightly, previous versions), architectures (e.g., CPU, GPU), etc. Therefore, to better show the detailed building results, developers use multiple badges. Second, many projects have multiple components, which may have different primary programming languages, different application scenarios (e.g., mobile, web, client, server), etc. They have different characteristics and developers may build them separately on different/the same platforms. Third, developers may build their project on different platforms simultaneously to get more comprehensive and accurate results, or conduct different kinds of analysis on different platforms at the same time. As most platforms can provide certain degree of free services, using multiple platforms is a good choice for developers.

Furthermore, the usage status of property combinations is to some extent decided by service providers. If one provider is enough for the properties that developers want to observe, all of its properties are taken into consideration. Otherwise, developers shall seek for other tools for compensation. In other words, in practice, more service providers are needed to satisfy the demand of developers.

Finding: The property combinations between Buildability and others (including Buildability itself) are the most frequent. It reflects the common practice to integrate CI platforms with other code quality evaluation services or adopt a comprehensive service which focus on multiple aspects. At the same time, developers also tend to use multiple CI platforms simultaneously.

Implications: For developers, it is convenient to adopt some new services given that they have used certain CI platforms. This could bring advantages of their project while requiring no much work of developers. When multiple CI platforms are used, the choices of new services are even more extensive. For badge providers, because code quality properties are related to the original repository and are updated along the code changes, it would be a good choice to integrate the tools with certain CI platforms to promote them.

4.3 Analysis in different programming languages

Besides previous analysis, due to the different characteristics and ecosystems in different programming languages, they are supposed to have different features when using QA badges. In this section, we conduct similar experiments to previous ones in each programming language, and aim to find their unique features.

We present the frequency of code quality properties in each programming language in Table 6. Note that the code quality properties are listed based on the descendent order of their occurrence.

Table 6 #Occurrences of code quality Properties.

Python				*C*			
Property name	#Occurrences	Property name	#Occurrences	Property name	#Occurrences	Property name	#Occurrences
Buildability	3,080 (91.58%)	Code Smells	229 (6.81%)	Buildability	1,702 (94.92%)	Documentation	80 (4.46%)
Code Coverage	1,423 (42.31%)	Security	211 (6.27%)	Security	325 (18.13%)	Code Style	62 (3.46%)
Documentation	776 (23.07%)	Reliability	156 (4.64%)	Reliability	279 (15.56%)	Code Smells	58 (3.23%)
Code Style	376 (11.18%)	License	42 (1.25%)	Code Coverage	253 (14.11%)	License	29 (1.62%)
JavaScript				*Java*			
Property name	#Occurrences	Property name	#Occurrences	Property name	#Occurrences	Property name	#Occurrences
Buildability	3,720 (94.32%)	Reliability	92 (2.33%)	Buildability	2,206 (93.99%)	Reliability	252 (10.74%)
Code Coverage	1,274 (32.30%)	License	69 (1.75%)	Code Coverage	624 (26.59%)	Code Style	206 (8.78%)
Code Style	338 (8.57%)	Documentation	48 (1.22%)	Security	300 (12.78%)	Documentation	30 (1.28%)
Code Smells	274 (6.95%)	Legality	3 (0.08%)	Code Smells	259 (11.04%)	License	28 (1.19%)
Security	207 (5.25%)						
TypeScript				*C#*			
Property name	#Occurrences	Property name	#Occurrences	Property name	#Occurrences	Property name	#Occurrences
Buildability	3,781 (91.86%)	Reliability	170 (4.13%)	Buildability	2,012 (95.45%)	Code Style	76 (3.61%)
Code Coverage	1,600 (38.87%)	License	78 (1.90%)	Code Coverage	319 (15.13%)	License	26 (1.23%)
Code Style	1,437 (34.91%)	Documentation	20 (0.49%)	Security	114 (5.41%)	Documentation	24 (1.14%)
Security	374 (9.09%)	Mutation Score	4 (0.10%)	Reliability	107 (5.08%)	Mutation Score	1 (0.05%)
Code Smells	346 (8.41%)			Code Smells	85 (4.03%)		
PHP				*C++*			
Property name	#Occurrences	Property name	#Occurrences	Property name	#Occurrences	Property name	#Occurrences
Buildability	3,554 (92.26%)	Security	354 (9.19%)	Buildability	2,388 (95.87%)	Code Style	158 (6.34%)
Code Coverage	1,799 (46.70%)	Legality	274 (7.11%)	Code Coverage	455 (18.27%)	Documentation	148 (5.94%)
Code Smells	1,100 (28.56%)	Documentation	51 (1.32%)	Security	340 (13.65%)	Code Smells	146 (5.86%)
Code Style	438 (11.37%)	License	32 (0.83%)	Reliability	295 (11.84%)	License	41 (1.65%)
Reliability	355 (9.22%)	Mutation Score	14 (0.36%)				
Shell				*Ruby*			
Property name	#Occurrences	Property name	#Occurrences	Property name	#Occurrences	Property name	#Occurrences
Buildability	1,225 (95.63%)	Security	33 (2.58%)	Buildability	3,818 (94.90%)	Security	96 (2.39%)
Code Coverage	66 (5.15%)	Reliability	31 (2.42%)	Code Coverage	1,867 (46.41%)	Code Style	95 (2.36%)
Code Style	59 (4.61%)	Documentation	24 (1.87%)	Code Smells	1,635 (40.64%)	Reliability	47 (1.17%)
Code Smells	46 (3.59%)	License	18 (1.41%)	Documentation	263 (6.54%)	License	12 (0.30%)

As we can see, Buildability and Code Coverage are the most frequent in almost all programming languages. Besides, the distribution of remaining properties differ a lot among all programming languages. Here, we take a look at *Python* and *C*.

For *Python*, 776 projects care about Documentation, which are much more than projects written in other languages. Taking *aio-libs/aiohttp* as an example, it is an asynchronous HTTP client/server framework and uses Read the Docs to write its documentation, which provides a badge docs passing showing the current build status of documentation. After inspecting the real cases, we summarize the reasons into the following aspects. First, many Python projects are packages hosted on PyPI, and most of them are supposed to be well documented according to the demand of PyPI. Second, most of these projects use Read the Docs to write documentation. However, other badges that are related to Documentation are few. Therefore, it "seems" that *Python* projects care more about Documentation. It is also reported that there has been a trend in the Python community to improve code quality by dictating "one right way" [16]. The maturity of the community and the effort of adhering to best practices may result in this scenario.

For C, compared to the overall results in Table 4, developers care about Security (vulnerabilities) much. In fact, according to many online discussions[7)][8)][9)], C is sometimes regarded as the most vulnerable programming language. Therefore, various tools aim to ensure the security of C and try to find possible vulnerabilities through static analysis or dynamic execution. Developers also pay attention to software security and take the usage of such tools.

Finding: Projects in different languages have different concerns on code quality properties. For example, projects written in *Python* care more about Documentation, while projects written in C care more about Security (vulnerabilities). One possible reason is different programming languages have different characteristics; therefore, developers tend to focus on different aspects. Another possible reason is the tool support situations in different programming languages differ, which means developers may be constrained by finite kinds of badges although they may have a demand for QA badges focusing on other properties.

Implications: Badge providers could consider designing QA badges related to properties that are less concerned at this time. The infrequent usage of these badges does not definitely indicate their unimportance. On the contrary, it may reflect the gap between existing QA badges and the requirements of developers – badge providers could facilitate the concern about various properties of developers and improve the completeness of all kinds of badges.

5 RQ3: Correlations between QA badges and software metrics

In RQ3, we conduct analyses **outside** QA badges, i.e., to see the relationship between QA badges and other metrics that are related to code quality, contribution quality, popularity, etc.

Ideally, we expect QA badges to have correlations with code-quality-related, contribution-quality-related, and popularity-related characteristics. Therefore, as we say in Section 2.2, we choose the following metrics that have potential relationship with code/contribution quality and popularity. Here, we also present how we expect the relationship between them looks like.

The number of bug-fixing commits reflects (1) the number of bugs and (2) the enthusiasm of developers to fix bugs in their software. The relationship between this metric and QA badges is two-fold. On one hand, QA badges may help developers improve code quality and bug-fixing commits are fewer. One the other hand, QA badges may accelerate the exposure of bugs and bug-fixing commits are more.

The number of bug issues reflects how many bugs have been confirmed by developers. The relationship between this metric and QA badges is also two-fold. On one hand, QA badges may help developers improve code quality and bug issues are fewer. On the other hand, QA badges may accelerate the exposure of bugs and bug issues are more.

Lines of Testing Code (TLOC) [4] reflects the amount of testing code in software, where more testing code indicates more efforts on quality assurance. Intuitively, QA badges may be positively correlated with lines of testing code.

The number of commits containing testing code [4] reflects the frequency of developers contributing to improve testing code, which indicates quality contribution. Intuitively, the usage of QA badges lead to more contribution on testing code.

The numbers of stars, forks, and watchers [4,16,18] reflect the software popularity, and we expect QA badges lead to more stars, forks, and watchers.

The above metrics are commonly used in previous work [4, 16, 18] and have potential relationship with code quality, contribution quality, popularity, etc.

Because there are many other key features that may have correlations with our metrics, following previous studies [16, 19, 20], we include the following variables as control variables: lines of code (LOC), number of commits, number of issues, number of contributors, and age. In our statistical analysis process, we regard them as covariates to obtain reliable analysis results.

7) https://medium.com/hackernoon/top-5-vulnerable-programming-languages-eab3144d6db7
8) https://thehackernews.com/2015/12/programming-language-security.html
9) https://www.digitalinformationworld.com/2019/03/searching-for-the-most-secure-programming-language.html

5.1 Analysis between projects with and without QA badges

In this section, we first compare the latest versions of open-source projects with and without QA badges. Specifically, we check whether they have differences on our concerned metrics that indicate correlations between QA badges and these metrics.

Because we cannot assume the relationship between QA badges and other metrics are linear, and other metrics may depend linearly on unknown smooth functions of some predictor variables [27], we take the use of Generalized Additive Models (GAM) [28] to perform the experiments. GAM provides a flexible and effective technique for modelling nonlinear relationships between variables, and parameters in GAM are estimated by a quadratically penalised likelihood type approach. Specifically, smooth terms are represented using penalized regression splines. At the same time, we also check for multicollinearity by calculating the variabce inflation factor (abbreviated as VIF) [29].

The experimental results are presented in Table 7, where for each metric, we present a subtable. The first row (treatment) represents whether QA badge(s) exist in one project, which is the variable we care about, and other rows are control variables. The columns represent the coefficient, standard error, t value, probability ($> |t|$), and significance, respectively. Concretely, the sign in the coefficient of treatment means it is positively/negatively correlated with the corresponding indicator, while significance shows whether the relationship is statistically significant. Note that for each statistical process, we check the multicollinearity and find low correlation between independent variables. Besides, we also compute the adjusted R-squared (R^2) which represents the deviance explained.

For *#Bug-fixing commits* and all other six metrics, the results are statistically significant ($p < 0.001$). Specifically, first, QA badges are negatively correlated with the number of bug-fixing commits and the number of bug issues ($R^2 = 0.884$ and 0.769), which indicates that projects having QA badges tend to have fewer bugs, i.e., higher code quality. For lines of testing code (TLOC), QA badges is positively correlated with it ($R^2 = 0.323$), which means although the model is not fully fitted, QA badges potentially positively encourage developers to write more testing code to better maintain their software. For the number of commits containing testing code, QA badges are positively correlated with it ($R^2 = 0.0892$), which means we cannot definitely conclude QA badges explain more contributions on testing code. Finally, for the numbers of stars, forks, and watchers, QA badges are also positively correlated with them ($R^2 = 0.197, 0.207, 0.155$), which means although the model do not fully fitted, the adoption of QA badges potentially have relationship with more people's attention and more popularity of their software.

Finding: Through GAM, we find that QA badges have statistically significant correlations with many software metrics. Specifically, QA badges are negatively correlated with bug-related metrics and positively correlated with testing-code-contribution-related metrics and popularity-related metrics.

5.2 Longitudinal analysis

Different from Section 5.1, in this section, we focus on projects that are using QA badges currently and compare the current versions with historical versions when QA badges have not been introduced.

Specifically, we first use the *git log* command and keywords of each QA badge we identified to find the commit in the history that firstly introduced QA badge(s), which is called *intro-commit* from now on. The key feature of *intro-commit* is there is no QA badge before it while at least one QA badge is added after it. Therefore, we compare the current version and previous version (the *parent* of *intro-commit*) and conduct correlation analysis.

We apply Generalized Additive Models (GAM) [28] and inspect the **correlation** between QA badges and other metrics. Concretely, the treatment variable is whether a project has introduced QA badges (1 for the current version while 0 for the previous version). Similar to the previous section, the metrics include the number of bug-fixing commits, the number of bug issues, lines of testing code, and the number of commits containing testing code. We also include *LOC*, *#Commits*, *#Issues*, *#Contributors*, and *Age* as control variables. Note that they are of vital significance here because the current version somehow "contains" the previous version. Besides, we also compute the adjusted R-squared (R^2) which represents the deviance explained.

The results are shown in Table 8, where the meaning of each row is the same as that in Table 7. Note that for each statistical process, we check the multicollinearity and find low correlation between independent variables. The coefficients of treatment variables are all statistically significant ($p < 0.001$) except for the analysis on #Bug issues ($p < 0.1$). Specifically, the adoption of QA badges is negatively correlated

Table 7 Statistical analysis results on QA badges for Section 5.1

#Bug-fixing commits

Variable	Coeff.	Std. Error	t value	Pr(>|t|)	Signif.
Treatment	-2.620e-05	1.426e-06	-18.37	<2e-16	***
LOC	3.694e-05	2.166e-06	17.05	<2e-16	***
#Commits	4.960e-02	1.227e-04	404.40	<2e-16	***
#Contributors	-1.953e-01	1.066e-02	-18.33	<2e-16	***
#Issues	3.223e+00	2.977e+00	1.083	0.279	
Age	-1.148e-07	5.023e-09	-22.86	<2e-16	***
(Intercept)	-6.436e-06	3.503e-07	-18.37	<2e-16	***

#Bug issues

Variable	Coeff.	Std. Error	t value	Pr(>|t|)	Signif.
Treatment	-4.016e-02	1.937e-03	-20.735	< 2e-16	***
LOC	6.136e-02	3.638e-04	168.657	< 2e-16	***
#Commits	2.154e-01	3.981e-02	5.412	6.29e-08	***
#Contributors	5.430e+01	2.619e+00	20.732	< 2e-16	***
#Issues	3.636e-02	1.535e-04	236.813	< 2e-16	***
Age	7.858e-06	1.207e-06	6.513	7.53e-11	***
(Intercept)	1.303e-03	6.285e-05	20.737	< 2e-16	***

Lines of Testing Code (TLOC)

Variable	Coeff.	Std. Error	t value	Pr(>|t|)	Signif.
Treatment	7.191e-03	2.662e-04	27.02	<2e-16	***
LOC	6.034e-02	4.042e-04	149.28	<2e-16	***
#Commits	-1.127e+00	2.289e-02	-49.24	<2e-16	***
#Contributors	5.373e+01	1.989e+00	27.02	<2e-16	***
#Issues	-1.788e-01	1.417e+00	-0.126	0.900	
Age	1.351e-05	9.373e-07	14.41	<2e-16	***
(Intercept)	1.766e-03	6.537e-05	27.02	<2e-16	***

#Commits containing testing code

Variable	Coeff.	Std. Error	t value	Pr(>|t|)	Signif.
Treatment	1.556e-04	8.979e-06	17.33	<2e-16	***
LOC	-2.184e-04	1.364e-05	-16.01	<2e-16	***
#Commits	2.775e-02	7.722e-04	35.94	<2e-16	***
#Contributors	1.163e+00	6.710e-02	17.33	<2e-16	***
#Issues	3.733e-07	2.064e-07	1.809	0.0705	.
Age	6.339e-07	3.162e-08	20.05	<2e-16	***
(Intercept)	3.822e-05	2.206e-06	17.33	<2e-16	***

#Stars

Variable	Coeff.	Std. Error	t value	Pr(>|t|)	Signif.
Treatment	2.739e-03	2.449e-05	111.86	< 2e-16	***
LOC	-2.916e-04	3.720e-05	-7.84	4.58e-15	***
#Commits	-1.350e-01	2.106e-03	-64.11	< 2e-16	***
#Contributors	2.047e+01	1.830e-01	111.86	< 2e-16	***
#Issues	7.847e+00	1.211e+01	0.648	0.516846	
Age	3.540e-06	8.624e-08	41.05	< 2e-16	***
(Intercept)	6.729e-04	6.015e-06	111.86	< 2e-16	***

#Forks

Variable	Coeff.	Std. Error	t value	Pr(>|t|)	Signif.
Treatment	7.128e-04	6.315e-06	112.866	< 2e-16	***
LOC	-7.822e-05	9.592e-06	-8.155	3.56e-16	***
#Commits	-3.300e-02	5.431e-04	-60.772	< 2e-16	***
#Contributors	5.326e+00	4.719e-02	112.868	< 2e-16	***
#Issues	5.239e+00	3.724e+00	1.407	0.159	
Age	6.614e-07	2.224e-08	29.742	< 2e-16	***
(Intercept)	1.751e-04	1.551e-06	112.866	< 2e-16	***

#Watchers

Variable	Coeff.	Std. Error	t value	Pr(>|t|)	Signif.
Treatment	9.174e-05	9.742e-07	94.178	< 2e-16	***
LOC	-7.846e-06	1.480e-06	-5.303	1.15e-07	***
#Commits	-4.207e-03	8.377e-05	-50.220	< 2e-16	***
#Contributors	6.856e-01	7.279e-03	94.180	< 2e-16	***
#Issues	7.149e-07	1.488e-06	0.481	0.631	
Age	2.054e-07	3.431e-09	59.874	< 2e-16	***
(Intercept)	2.254e-05	2.393e-07	94.178	< 2e-16	***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1

with the number of bug-fixing commits ($R^2 = 0.56$), which is consistent with results in Section 5.1, and it indicates the introduction of QA badges may have potential relationship with the decrease of bugs. The adoption of QA badges is also negatively correlated with the number of bug issues although it is only significant at a level of 0.1 ($R^2 = 0.49$). Besides, the adoption of QA badges also has "potential" impacts on lines of testing code and the number of commits containing testing code ($R^2 = 0.442$ and 0.277).

Finding: According to the longitudinal analysis, the adoption of QA badges is correlated with the number of bug-fixing commits (negative), lines of testing code (positive), and the number of commits containing testing code (positive).

6 Threats to Validity

The threats to internal validity lie in the implementation. To mitigate this threat, the first two authors independently reviewed the experimental scripts to check their correctness.

The threats to external validity lie in the subjects used in this study. We use popular projects in GitHub, which may not be representative of all projects and hampers the generalizability of our observations. To mitigate this threat, we select a substantial number of top-popular projects (i.e., top-10,000) like previous work [16,20,30] for 10 popular programming languages, based on which we believe this study can provide practical findings for the community. Besides, the collection date of our data is Aug 5, 2021,

Table 8 Longitudinal analysis results on QA badges for Section 5.2.

#Bug-fixing commits					
Variable	Coeff.	Std. Error	t value	Pr(>|t|)	Signif.
Treatment	-2.314e-08	9.766e-11	-236.976	$< 2e-16$	***
LOC	6.972e-06	1.633e-06	4.269	1.97e-05	***
#Commits	3.636e-02	1.534e-04	236.979	$< 2e-16$	***
Age	-8.257e-08	5.749e-09	-14.363	$< 2e-16$	***
#Issues	-5.038e-07	2.840e-07	-1.774	0.0761	.
(Intercept)	1.280e-07	5.403e-10	236.981	$< 2e-16$	***

#Bug issues					
Variable	Coeff.	Std. Error	t value	Pr(>|t|)	Signif.
Treatment	-2.163e+01	1.208e+01	-1.791	0.0734	.
LOC	-1.251e-04	1.265e-05	-9.888	$< 2e-16$	***
#Commits	8.914e-02	6.953e-04	128.210	$< 2e-16$	***
Age	1.527e-07	7.939e-08	1.923	0.0545	.
#Issues	3.851e-09	9.165e-12	420.160	<2e-16	***
(Intercept)	7.306e+01	1.857e+01	3.934	8.37e-05	***

Lines of Testing Code (TLOC)					
Variable	Coeff.	Std. Error	t value	Pr(>|t|)	Signif.
Treatment	3.431e-07	1.587e-08	21.62	<2e-16	***
LOC	4.854e-02	2.655e-04	182.86	<2e-16	***
#Commits	5.393e-01	2.494e-02	21.63	<2e-16	***
Age	1.739e-05	9.343e-07	18.61	<2e-16	***
#Issues	3.728e+00	2.894e-02	128.79	<2e-16	***
(Intercept)	1.900e-06	8.780e-08	21.64	<2e-16	***

#Commits containing testing code					
Variable	Coeff.	Std. Error	t value	Pr(>|t|)	Signif.
Treatment	4.557e-08	3.539e-10	128.794	<2e-16	***
LOC	-6.367e-06	5.918e-06	-1.076	0.282	
#Commits	7.161e-02	5.560e-04	128.795	<2e-16	***
Age	4.890e-07	2.083e-08	23.478	<2e-16	***
#Issues	1.153e-04	8.951e-07	128.79	<2e-16	***
(Intercept)	2.521e-07	1.957e-09	128.796	<2e-16	***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1

which is also not quite new. However, our dataset is still large enough to obtain convincing findings and implications.

The threats to construct validity lie in badge identification, category construction, and statistical analysis. (1) *QA badges identification.* According to Section 2.2, we automatically extract badges based on pre-identified "img" tags and only consider the tags whose frequency is at least 10, which may miss some badges in the dataset. To mitigate this issue, we randomly select 500 tags that occur less than 10 times and find most of them (i.e., 93.8%) are not QA badges after our manual inspection. Therefore, our procedure can minimize the threat while reducing manual efforts. (2) *Category construction.* To reduce threats caused by subjectivity in taxonomy construction, we follow an open-coding procedure [14] and involve multiple participators in manual labelling. Moreover, the inter-rater agreement is relatively high (0.863), demonstrating the high reliability of the procedure. (3) *Statistical analysis.* To mitigate the threats from indicators and statistical analysis, we adopt indicators widely used in previous work [4,16,18] with potential relationship with code quality, contribution quality, and popularity. Moreover, we consider multiple variables as control variables [16, 19, 20] to avoid the impact of unobserved variables. Moreover, the R^2 values can also show the fitness.

7 Related Work

In this section, we summarize related work to situate our work within the literature.

Study on software QA. There exists a large body of work related to the software QA status. Rai et al. [31] conducted an extensive review of the literature to find popular areas that receive attention from researchers. Zhao et al. [1] explored how software QA is performed under the open-source model. They found that open-source has certainly introduced a new dimension in large-scale distributed software development. Holck et al. [32] investigated the use of continuous integration in FreeBSD and Mozilla and found it challenging to balance the access required to add contributions against the need to stabilize and mature the software before a release. Khanjani et al. [33] reviewed the literature on QA under OSS development methods and techniques. The results show the process of QA of OSS and how it can affect the overall QA principle. Bahamdain [3] discussed the stakeholders of the OSS community, the QA frameworks and models, the problems that affect the quality, and the advantages and disadvantages of OSS compared to closed-source software. Axelsson et al. [34] investigated the challenges related to QA in software ecosystems and identified what approaches had been proposed in the literature. Hassan et al. [35] reviewed several QA techniques that aim to improve software quality. They found that these techniques play an important role in software quality improvement, while some vulnerabilities were also found. Ma et al. [2] performed a large-scale study on QA, security, and interpretation of deep learning and pinpointed challenges and future opportunities. Felderer et al. [36] targeted the QA of AI-based systems. They defined basic concepts of AI-based systems and characterized their artifact type, process, and quality characteristics. They also identified eight key challenges of AI-based systems. Inspired by these previous studies, we aim to understand the QA badges in OSS, which could look at the QA situation from a new perspective to some extent. To the best of our knowledge, we are the first authors to conduct such a study.

Study on badges. Many studies have examined software ecosystems in different aspects, including communication [37–40], changes [41–43], dependencies [44–47], static analysis [48], testing and continuous integration [49–51]. However, there exists little work focusing on badges. Trockman et al. [4] conducted an empirical study of badges in npm packages. They found that non-trivial badges, which display the build status, test coverage, and currency of dependencies, are mostly reliable signals, correlating with more tests, better pull requests, and fresher dependencies. Legay et al. [10] analyzed repositories in Cargo and Packagist and found that the most widespread badges convey either static information or relay information about the build status of a project and are typically added early in projects. Inspired by previous work, especially Trockman et al. [4], our study focuses on QA badges which make up a large proportion of all badges, and presented a detailed analysis of them. Also, our dataset is more general and representative, making our findings more reliable and generalizable.

8 Conclusion

In this paper, we present the first large-scale empirical study on QA badges on GitHub. We collect a large dataset containing 100,000 popular GitHub projects written in 10 popular programming languages and their 57,363 badges. Inside QA badges, we build a category for all QA badges based on the properties they focus on, which shows the types of QA badges developers use. Then, we analyze the frequency of the properties that QA badges focus on, and property combinations, too, which presents their use status. Outside QA badges, we conduct a correlation analysis between QA badges and some software metrics that have potential relationships with code quality, contribution quality, and popularity. We find that QA badges have statistically significant correlations with many software metrics.

Acknowledgements This work is partly supported by the National Natural Science Foundation of China under Grant No. 61872008.

Appendix A (1) GitHub: https://github.com/ (2) Bitbucket: https://bitbucket.org/ (3) Travis CI: https://travis-ci.org/ (4) Coveralls: https://coveralls.io/ (5) Snyk: https://snyk.io/ (6) DeepScan: https://deepscan.io/ (7) Code Climate: https://codeclimate.com/ (8) Codacy: https://www.codacy.com/ (9) SymfonyInsight: https://insight.symfony.com/ (10) Stryker Mutator: https://stryker-mutator.io/ (11) Inch-CI: http://inch-ci.org/ (12) Read the Docs: https://readthedocs.org/ (13) FOSSA: https://fossa.com/ (14) CII Best Practice: https://bestpractices.coreinfrastructure.org/ (15) SonarCloud: https://sonarcloud.io/ (16) Scrutinizer: https://scrutinizer-ci.com/ (17) StyleCI: https://styleci.io/ (18) Prettier: https://prettier.io/ (19) HoundCI: https://houndci.com/ (20) elasticsearch-sql: https://github.com/NLPchina/elasticsearch-sql (21) PyPI: https://pypi.org/ (22) aiohttp: https://github.com/aio-libs/aiohttp

References

1 Zhao L, Elbaum S. Quality assurance under the open source development model. Journal of Systems and Software, 66(1):65–75, 2003
2 Ma L, Juefei-Xu F, Xue M, et al. Secure deep learning engineering: A software quality assurance perspective. arXiv preprint arXiv:181004538, 2018
3 Bahamdain S S. Open source software (oss) quality assurance: A survey paper. Procedia Computer Science, 56:459–464, 2015
4 Trockman A, Zhou S, Kästner C, et al. Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In: Proceedings of the 40th International Conference on Software Engineering, pages 511–522. 2018
5 Walkinshaw N. Software quality assurance. Springer International Publishing, 10:978–3, 2017
6 Laporte C Y, April A. Software quality assurance. John Wiley & Sons, 2018
7 Perera P, Silva R, Perera I. Improve software quality through practicing devops. In: 2017 Seventeenth International Conference on Advances in ICT for Emerging Regions (ICTer), pages 1–6. IEEE, 2017
8 Basu A. Software quality assurance, testing and metrics. PHI Learning Pvt. Ltd., 2015
9 Wong W E. Special section on software quality assurance: research and practice. IEEE Transactions on Reliability, 65(1):3–3, 2016
10 Legay D, Decan A, Mens T. On the usage of badges in open source packages on github. In: BENEVOL. 2019
11 Parsons V L. Stratified sampling. Wiley StatsRef: Statistics Reference Online, pages 1–11, 2014
12 Avelino G, Constantinou E, Valente M T, et al. On the abandonment and survival of open source projects: An empirical investigation. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pages 1–12. IEEE, 2019
13 Steinmacher I, Pinto G, Wiese I S, et al. Almost there: A study on quasi-contributors in open-source software projects. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pages 256–266. IEEE, 2018
14 Seaman C B. Qualitative methods in empirical studies of software engineering. IEEE Transactions on software engineering, 25(4):557–572, 1999
15 Landis J R, Koch G G. The measurement of observer agreement for categorical data. Biometrics, pages 159–174, 1977
16 Zhang J, Li F, Hao D, et al. A study of programming languages and their bug resolution characteristics. IEEE Transactions on Software Engineering, 2019

17 Berger E D, Hollenbeck C, Maj P, et al. On the impact of programming languages on code quality: a reproduction study. ACM Transactions on Programming Languages and Systems (TOPLAS), 41(4):1–24, 2019

18 Borges H, Valente M T. What's in a github star? understanding repository starring practices in a social coding platform. Journal of Systems and Software, 146:112–129, 2018

19 Ortu M, Marchesi M, Tonelli R. Empirical analysis of affect of merged issues on github. In: 2019 IEEE/ACM 4th International Workshop on Emotion Awareness in Software Engineering (SEmotion), pages 46–48. IEEE, 2019

20 Ray B, Posnett D, Filkov V, et al. A large scale study of programming languages and code quality in github. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 155–165. 2014

21 ISO Central Secretary. Systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models. Standard ISO/IEC 25010:2011, International Organization for Standardization, Geneva, CH, 2011

22 Fischer-Nielsen A, Fu Z, Su T, et al. The forgotten case of the dependency bugs: On the example of the robot operating system. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pages 21–30. IEEE, 2020

23 Asadollah S A, Sundmark D, Eldh S, et al. Concurrency bugs in open source software: a case study. Journal of Internet Services and Applications, 8(1):1–15, 2017

24 Jin G, Song L, Shi X, et al. Understanding and detecting real-world performance bugs. ACM SIGPLAN Notices, 47(6):77–88, 2012

25 Chilenski J J, Miller S P. Applicability of modified condition/decision coverage to software testing. Software Engineering Journal, 9(5):193–200, 1994

26 Zhao Y, Feng Y, Wang Y, et al. Quality assessment of crowdsourced test cases. Science China Information Sciences, 63(9):1–16, 2020

27 Stasinopoulos D M, Rigby R A, et al. Generalized additive models for location scale and shape (gamlss) in r. Journal of Statistical Software, 23(7):1–46, 2007

28 Hastie T J, Tibshirani R J. Generalized additive models. Routledge, 2017

29 Mansfield E R, Helms B P. Detecting multicollinearity. The American Statistician, 36(3a):158–160, 1982

30 Liu W, Chen B, Peng X, et al. Identifying change patterns of api misuses from code changes. Science China Information Sciences, 64(3):1–19, 2021

31 Rai A, Song H, Troutt M. Software quality assurance: An analytical survey and research prioritization. Journal of Systems and Software, 40(1):67–83, 1998

32 Holck J, Jørgensen N, et al. Continuous integration and quality assurance: A case study of two open source projects. Australasian Journal of Information Systems, 11(1), 2003

33 Khanjani A, Sulaiman R. The process of quality assurance under open source software development. In: 2011 IEEE Symposium on Computers & Informatics, pages 548–552. IEEE, 2011

34 Axelsson J, Skoglund M. Quality assurance in software ecosystems: A systematic literature mapping and research agenda. Journal of Systems and Software, 114:69–81, 2016

35 Hassan M U, Mubashir M, Shabir M A, et al. Software quality assurance techniques: A review. International Journal of Information, Business and Management, 10(4):214–221, 2018

36 Felderer M, Ramler R. Quality assurance for ai-based systems: Overview and challenges (introduction to interactive session). In: International Conference on Software Quality, pages 33–42. Springer, 2021

37 Bird C, Gourley A, Devanbu P, et al. Mining email social networks. In: Proceedings of the 2006 international workshop on Mining software repositories, pages 137–143. 2006

38 Guzzi A, Bacchelli A, Lanza M, et al. Communication in open source software development mailing lists. In: 2013 10th Working Conference on Mining Software Repositories (MSR), pages 277–286. IEEE, 2013

39 Joblin M, Apel S, Hunsen C, et al. Classifying developers into core and peripheral: An empirical study on count and network metrics. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 164–174. IEEE, 2017

40 Singer L, Figueira Filho F, Storey M A. Software engineering at the speed of light: how developers stay current using twitter. In: Proceedings of the 36th International Conference on Software Engineering, pages 211–221. 2014

41 Bogart C, Kästner C, Herbsleb J, et al. How to break an api: cost negotiation and community values in three software ecosystems. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 109–120. 2016

42 Decan A, Mens T, Claes M. An empirical comparison of dependency issues in oss packaging ecosystems. In: 2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER), pages 2–12. IEEE, 2017

43 Raemaekers S, Van Deursen A, Visser J. Semantic versioning versus breaking changes: A study of the maven repository. In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, pages 215–224. IEEE, 2014

44 Bavota G, Canfora G, Di Penta M, et al. How the apache community upgrades dependencies: an evolutionary study. Empirical Software Engineering, 20(5):1275–1317, 2015

45 Cox J, Bouwers E, Van Eekelen M, et al. Measuring dependency freshness in software systems. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, volume 2, pages 109–118. IEEE, 2015

46 Kula R G, German D M, Ouni A, et al. Do developers update their library dependencies? Empirical Software Engineering, 23(1):384–417, 2018

47 Mirhosseini S, Parnin C. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 84–94. IEEE, 2017

48 Zampetti F, Scalabrino S, Oliveto R, et al. How open source projects use static code analysis tools in continuous integration pipelines. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pages 334–344. IEEE, 2017

49 Hilton M, Tunnell T, Huang K, et al. Usage, costs, and benefits of continuous integration in open-source projects. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 426–437. IEEE, 2016

50 Vasilescu B, Yu Y, Wang H, et al. Quality and productivity outcomes relating to continuous integration in github. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pages 805–816. 2015

51 Zhao Y, Serebrenik A, Zhou Y, et al. The impact of continuous integration on other software development practices: a large-scale empirical study. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 60–71. IEEE, 2017