How Open Source Projects use Static Code Analysis Tools in Continuous Integration Pipelines

Fiorella Zampetti*, Simone Scalabrino†, Rocco Oliveto†, Gerardo Canfora*, Massimiliano Di Penta*
**University of Sannio, Italy* — †*University of Molise, Italy*

Abstract—**Static analysis tools are often used by software developers to entail early detection of potential faults, vulnerabilities, code smells, or to assess the source code adherence to coding standards and guidelines. Also, their adoption within Continuous Integration (CI) pipelines has been advocated by researchers and practitioners. This paper studies the usage of static analysis tools in 20 Java open source projects hosted on GitHub and using Travis CI as continuous integration infrastructure. Specifically, we investigate (i) which tools are being used and how they are configured for the CI, (ii) what types of issues make the build fail or raise warnings, and (iii) whether, how, and after how long are broken builds and warnings resolved. Results indicate that in the analyzed projects build breakages due to static analysis tools are mainly related to adherence to coding standards, and there is also some attention to missing licenses. Build failures related to tools identifying potential bugs or vulnerabilities occur less frequently, and in some cases such tools are activated in a "softer" mode, without making the build fail. Also, the study reveals that build breakages due to static analysis tools are quickly fixed by actually solving the problem, rather than by disabling the warning, and are often properly documented.**

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

I. Introduction

Initially advocated by Booch in the early nineties [16] and then introduced as part of the extreme programming practices [12], Continuous Integration (CI) is a software development practice foreseeing frequent builds integrations per day, then verified by an automated build machinery often hosted on a separate server. The automated verification performed by CI servers favors an early discover of integration errors without leaving the burden to developers, hence reducing effort [19]. Over the years, researchers have investigated the CI adoption [21], [25], [27], [33], and studied specific build phases, namely compilation [10] and testing [14].

Besides the usefulness of automated compilation and testing, software projects can greatly benefit of the execution of Automated Static Code Analysis Tools (ASCATs) within CI. For example, build breakages due to ASCATs checks can be used to force developers to follow coding standards/guidelines (Duvall *et al.* [19], p. 58) or to make sure source code is kept conform with the intended architecture as the project evolves (Duvall *et al.* [19], p. 59).

Previous work has studied the usage of ASCATs in software projects [23], analyzing how such tools were used to detect real faults [32], [39], [41], how the raised warnings are addressed [26], and whether and why there is an under usage of ASCATs in open source projects [13]. In particular, Beller *et al.* [13] found that (i) most projects rely on one ASCAT tool only, while very few use multiple ASCATs; and (ii) ASCATs are carefully configured by developers which enable or suppress some checks, while rarely introducing customized ones. They claim such a configuration would be particularly crucial when ASCATs are fully integrated in the development workflow, and in particular in a CI process. However, to the best of our knowledge, previous studies do not provide much details on how ASCATs are being used within a CI pipeline. While the traditional usage of ASCATs—*i.e.,* within private builds—is left to the burden of the single developer and does not impact the others, the introduction of such tools in a CI pipeline can have positive effects (automated and consistent checks, better awareness of coding standards) but also negative effects (unnecessary, annoying build breakages) on the development process [19].

The goal of this paper is to investigate ASCATs usage practices within CI pipelines. More specifically, the paper studies the use of ASCATs within CI pipelines of 20 popular Java open source projects hosted on GitHub and using Travis CI to support CI activities[1]. We start our study by analyzing which ASCATs are used by the studied projects, and how they have been configured to run within a CI pipeline. Then, we look into what types of issues, raised by the tools, cause build breakages or are simply reported as warnings. Last, but not least, we investigate whether, after how long, and how build breakages due to static analysis are resolved.

The obtained results show cases where ASCATs are configured to fail the builds, especially for checks (*e.g.,* related to coding standard adherence) developers consider important and do not typically produce false positives (*e.g.,* unused imports or formatting issues, as opposed to likely defects). Also, the results provide insights on what are the typical kinds of checks that make the build fail, and how such builds are fixed. Based on such results, we provide a set of recommendations to developers for a more effective and efficient usage of ASCATs within CI.

II. Empirical Study Definition and Planning

The *goal* of this study is to analyze the CI process of open source Java projects, with the *purpose* of understanding the usage of ASCATs within a CI pipeline.

[1]The study working datasets are available in a replication package [40].

TABLE I
PROJECTS ANALYZED IN THE STUDY. THE SECOND COLUMN INDICATES THE BUILD FRAMEWORK (G)RADLE OR (M)AVEN.

Project Name	Build Framework	# of Stars	Time Interval Analyzed	# of Builds	# of Broken Builds (%)
RxJava	G	19,673	08-2014/09-2015	391	46 (11.76%)
retrofit	M	17,598	10-2012/09-2015	558	43 (7.71%)
okhttp	M	16,198	10-2012/09-2015	1,436	732 (50.97%)
glide	G	12,273	06-2014/11-2014	108	0
picasso	M	12,183	01-2014/12-2014	205	27 (13.17%)
zxing	M	11,439	01-2014/08-2015	296	17 (5.74%)
dagger	M	5,798	10-2012/09-2014	250	25 (10.00%)
dropwizard	M	5,258	04-2013/09-2015	633	57 (9.01%)
metrics	M	4,488	04-2013/08-2015	295	30 (10.17%)
auto	M	4,133	07-2013/09-2015	181	34 (18.78%)
roboguice	M	3,885	02-2013/12-2014	200	68 (34.00%)
checkstyle	M	1,545	09-2013/09-2015	1,486	97 (6.53%)
morphia	G	1,041	12-2013/08-2015	272	27 (9.93%)
springfox	G	1,030	05-2013/05-2015	505	58 (11.49%)
BuildCraft	G	901	05-2014/09-2015	1,273	628 (49.33%)
ElasticSearch-Hadoop (ESH)	G	890	05-2013/08-2015	736	31 (4.21%)
SpongeAPI	G	843	09-2014/09-2015	880	90 (10.23%)
embulk	G	786	02-2015/09-2015	457	34 (7.44%)
mifosx	G	163	02-2013/09-2015	1,756	95 (5.41%)
griffon	G	160	03-2014/09-2015	217	60 (27.65%)

A. Study context and research questions

The *context* consists of 20 open source projects written in Java, hosted on GitHub that have a build process which explicitly specifies dependencies on ASCATs, and that adopt CI through the Travis CI infrastructure.

The 20 projects have been selected as follows. First, we identified the Java projects on GitHub using Travis CI, by looking at the *TravisTorrent*[2] dataset [15]. Then, we ranked the selected projects in terms of popularity[3] by using the GitHub APIs. After that, we filtered out all projects which do not use ASCATs by analyzing the build files. Finally, we balanced our dataset in terms of build tool by including 10 projects using Maven [1] and 10 using Gradle [6]. Note that we did not balance on the ASCATs as we also want to have an idea of the tools used more or less frequently in our sample. Table I lists the analyzed projects indicating their name on GitHub, the build framework, the number of stars they had on GitHub at the time they have been chosen, the analyzed time period, the total number of builds (we considered the set of builds TravisTorrent stored, for such projects, in November 2016), and the total number of broken builds.

The study aims at addressing the following research questions:

- **RQ$_0$:** *To what extent the analyzed projects use ASCATs?* This research question is preliminary to the core of the study and aims at determining what ASCATs are being used and, overall, how many builds fail or raise warnings because of ASCATs.
- **RQ$_1$:** *How are ASCATs configured in the CI process?* We analyze how ASCATs are configured within CI, *i.e.,* to break builds or to simply raise warnings. Also, as also done by Beller *et al.* [13], we analyze what kinds of checks have been enabled/disabled and whether they change over time.
- **RQ$_2$:** *What types of issues make build warn/fail?* Here we analyze the proportions of different kinds of warnings highlighted by ASCATs that cause build breakages or warned builds; out of the builds for which such warnings have been enabled (as per **RQ$_1$:**), we analyze how many result as broken or warned.
- **RQ$_3$:** *When and how are broken/warned builds resolved?* First, we analyze how long broken builds last, or warnings remain active. Then, we investigate what type of files (*i.e.,* source code, build scripts, CI configuration, or ASCAT configuration) were changed to induce a build failures, and types of files that were changed to make the build pass. Finally, we look at evidence of ASCAT-related documented build fixes.

[2]http://travistorrent.testroots.org

[3]The popularity is evaluated counting the number of stars related to each project of interest.

B. Data extraction process

To extract data needed in our analysis, we first analyzed Travis CI builds and job logs. After that, we combined this information with the projects' change history mined from the Git versioning system hosted on GitHub. Then, in order to identify the warnings produced by ASCATs, when they were not highlighted in the job logs, we had to locally build each snapshot for which the build failed or raised warnings.

1) Analysis of Travis CI builds and jobs: Travis CI provides a default build environment and a default set of steps, for each programming language, configurable in a file named `.travis.yml`. More specifically, a Travis CI build consists of two steps: *install*, which retrieves and installs the needed dependencies, and *script*, which runs the build script. It is possible to execute other commands before, between and after the *install* and *script* phase. The status of a build is computed considering the status of each single command executed. More in detail, if one of the commands executed before the *install* phase ends with a non-zero exit code, the build is immediately stopped and is marked as *errored*. Instead, if one of the script commands returns a non-zero exit code the build continues to run before being marked as failed. All commands executed after the script step do not influence the build status. Using the Travis CI APIs, we downloaded and extracted information about each build, and specifically the build status (passed, failed, errored or canceled), the commit ID related to the event, along with its message, timestamp, and the number of build jobs with their identifiers. Note that in this study we treat failed and errored builds together as both can contain failed jobs (errored are taken into account if they contained at least a failed job). Canceled builds are discarded. Then, for each job we downloaded its log produced during the build process. Each job log reports the output of each step defined in the build configuration file. Then, we use regular expressions[4] to (i) determine whether the build has produced any warning because of ASCATs (and in that case we marked the build as "warned") or whether the build has failed because of ASCATs; and (ii) when possible, to determine the types of warnings raised by ASCATs. When such information were not available, we needed to rerun the build locally as detailed in Section II-B4.

[4]Details in our replication package.

2) Integration with change information from GitHub: To analyze build and configuration files in **RQ**$_1$ and to address **RQ**$_3$, we mined the commit history of each project in our dataset. We considered (i) changes occurred to source code files; (ii) changes occurred to the configuration files of the ASCATs adopted, like `checkstyle.xml` or `findbugs-exclude.xml`; (iii) changes to build scripts, *i.e.,* `build.gradle` or `pom.xml`, and (iv) changes to the CI configuration files, *i.e.,* `.travis.yml`. In addition, we mined and manually analyzed all the commit messages related to builds that fixed sequences of ASCAT-related broken or warned builds.

3) Analysis of static analysis tools configurations: To address **RQ**$_1$, we extracted from the commit history of each project the whole set of commits in which there is at least one change made to the ASCAT configuration files, and to the build scripts (as ASCATs can be enabled/disabled from there). Then, for such commits, we checked out the file and analyzed its content. In order to analyze the XML configuration files, we parsed such files extracting information about the activated checks (for CheckStyle and PMD) and the suppressed patterns of checks and categories of checks (for both CheckStyle and FindBugs). We derived the categories for the specific checks using the catalogs provided on the CheckStyle and FindBugs websites. For each project, we extracted the macro-categories of checks activated through the configuration file at least once in the whole history of the project. Then, we analyzed how categories of checks are activated or deactivated throughout the history of each project. Finally, we extracted information about the practice of warning suppression, *i.e.,* the exclusion of some files from the check of specific warnings.

4) Rebuild and local execution of ASCATs: To address **RQ**$_2$, whenever possible, we relied on the information available in Travis CI build logs. By using regular expressions, we identified the types of warnings raised by the ASCATs. However, in some cases—and in particular for FindBugs and PMD—the tools could be configured so to only output the warnings in a separate XML/HTML file, and not in the build log. In such cases, we needed to locally reproduce the build performed by the CI infrastructure. To this aim, we cloned the project, checked out the snapshot in which the build was warned or errored, and rebuilt the project. More in detail, it is first necessary to analyze the content of the `.travis.yml`, in order to recognize the environment, *e.g.,* the Java version, and the script commands to be executed. After that, we set the environment on our machine and tried to execute the whole set of script commands needed for the ASCATs evaluation. In some cases, we needed to manually resolve some dependencies no longer available. After having locally rebuilt the project, we extracted and analyzed (using XML/HTML parsers) the content of the XML/HTML files generated by the ASCAT to determine the types of warnings raised. We then grouped warnings in macro-categories, following the official documentation of each ASCAT, mapping each check into its specific macro-category. As an example, Checkstyle foresees 14 check categories containing different checks, *e.g.,* the *Imports* category includes the *UnusedImports* check.

TABLE II
ASCAT USED BY THE ANALYZED SYSTEMS AND WHETHER THEY ARE CONFIGURED TO (B)REAK BUILDS OR RAISE (W)ARNINGS.

			ASCATs						
System	**Broken Builds for ASCATs**	**Warned Builds for ASCATs**	**CheckStyle**	**FindBugs**	**PMD**	**License**	**Apache-rat**	**Clirr**	**jDepend**
RxJava	0	373 (95.40%)		w		w			
retrofit	4 (0.72%)	0	b						
okhttp	33 (2.30%)	0	b						
glide	0	0	b						
picasso	2 (0.98%)	0	b						
zxing	2 (0.68%)	0	b	b			b	b	
dagger	1 (0.40%)	0	b						
dropwizard	6 (0.95%)	0		b					
metrics	1 (0.34%)	0		b					
auto	3 (1.66%)	16 (8.84%)	b/w						
roboguice	0	0	b	b	b				
checkstyle	0	0		b	b				
morphia	14 (5.15%)	0	b	b					
springfox	6 (1.19%)	0	b	b	b				
BuildCraft	444 (34.88%)	0	b						
ESH	0	139 (18.89%)		w	w				
SpongeAPI	26 (2.95%)	483 (54.89%)	w			b			
embulk	0	445 (97.37%)		w					
mifosx	56 (3.19%)	0				b			
griffon	13 (5.99%)	202 (93.09%)	b	w		w			w

III. Empirical Study Results

This section reports the study results, addressing the research questions formulated in Section II.

A. *RQ$_0$: To what extent the analyzed projects use ASCATs?*

Overall, the studied projects use seven different ASCATs (see Table II):

- CheckStyle [3], used in 13 projects, is typically used to check the adherence to coding standards, but also to identify pieces of code that are good candidates for code smells;
- FindBugs [5], used in 11 projects, is an ASCAT that aims at finding issues in the source code that can either result in faults or, in other cases, can hinder source code maintainability, security, or performance.
- PMD [9], used in four projects, is a source code analyzer able to find common programming flaws such unused objects, unnecessary `catch` blocks, or incomprehensible naming.
- License-gradle-plugin (hereby referred as "License") [8], used in four projects, has the purpose to scan source code files and determine whether their header match a given file, *e.g.,* a `LICENSE.txt` file containing the license text. Also, the tool is capable to add such a header when it is missing.
- Apache-rat [2], used by *zxing*, is a release auditing tool mainly focused on checking the occurrence of licenses in project versions.
- Clirr [4], used again only by *zxing*, aims at identifying compatibility problems between two artifacts, *e.g.,* two project versions or two different API implementations.
- jDepend [7], used by *griffon*, is a dependency analysis tool.

The results also indicate that 11 out of 20 projects use just one ASCAT, while the remaining rely on multiple tools. This is consistent with what previously found by Beller *et al.* [13]: on 122 repositories analyzed 36% of them use only a single ASCAT compared to 23% of them which, instead, use multiple ASCATs usually covering more than one programming language. In our case, while we focus on tools working on one programming language, different tools are likely used because they cover different kinds of issues.

When looking at the overall number and percentage of broken and warned builds due to ASCATs (second and third column of Table II), it can be noticed that except for one project (*BuildCraft*) the percentage of broken builds is between 0% and 6%. Warned builds—that obviously do not cause CI pipeline interruption—in some cases can get close to 100%. These percentages need a deeper analysis in the following RQs, to investigate whether they depend on the projects' code quality or on ASCATs' configuration.

RQ_0 summary. The 20 projects use seven ASCATs, and in 11 cases each project use one ASCAT only. ASCATs cause up to 6% of broken builds, and up to 97% of warned builds.

B. RQ_1: How are ASCATs configured in the CI process?

The right-side columns of Table II indicate whether ASCATs have been configured to break the build (b), to simply raise warnings (w), or both (b/w). Except for *SpongeAPI*, where CheckStyle was configured to simply produce warnings, and for *auto*, where the tool was changed from producing build failures to simply raise warnings during the observed history, the remaining projects configured CheckStyle to cause build breakages. FindBugs, instead, was in seven cases configured to break the build, while in the remaining four to just raise warnings. PMD was always configured to break the build, except for *ESH*, while License was configured to break the build in two projects, and in others two to simply raise warnings. Moreover, *zxing* uses both Apache-rat and Clirr in build breakage mode. Finally, *griffon* uses jDepend in warning mode.

Among the 20 projects taken into account, 17 used a customized configuration for at least one ASCAT. The project that did not configure ASCATs are *ESH*, *embulk* and *RxJava*. Also, the only configured ASCATs are CheckStyle, FindBugs, and PMD. Fig. 1 shows the percentage of checks (out of those available in the default configuration) that, for each tools, have been activated in the studied projects. Among 154 checks provided by CheckStyle, the percentage of checks activated is often low, always below 40%. FindBugs provides a greater number of checks (424), but in our dataset, only in two projects (*dropwizard* and *zxing*) it has some warnings suppressed for the whole projects. Finally, PMD provides 239 checks and only the *checkstyle* project has about half of them suppressed.

1) Detailed analysis of activated/suppressed checks: In the following, we report details about the configuration of the different ASCATs.

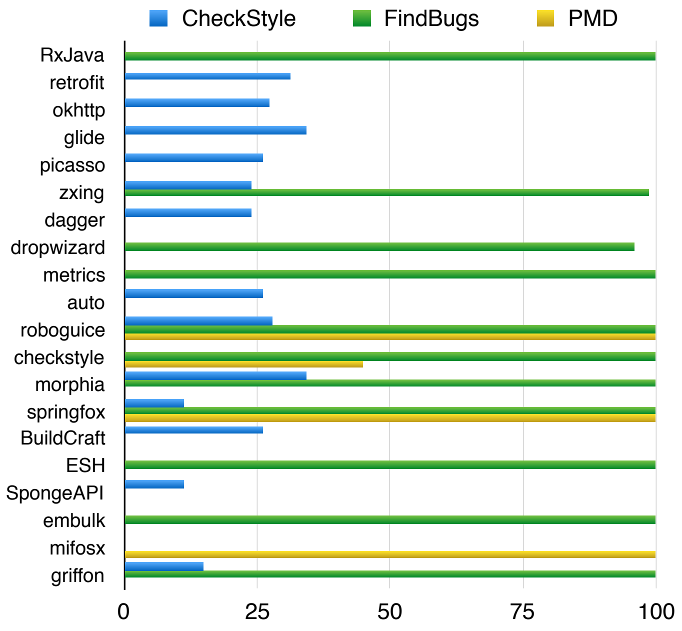

Fig. 1. Percentage of checks activated at least once.

CheckStyle. The configuration has been customized in all 13 projects using the ASCAT. Fig. 2 shows, for each category of CheckStyle checks, in how many projects there is at least one check belonging to such a category activated in the configuration file. For the sake of space, and to give a broad picture of what was activated, we did not report results at the granularity of single check. The most frequently activated categories are *Whitespace* (related to source code indentation), *Imports* (*e.g.,* checking unused imports), *Coding* (checking for possible defects) and *Block checks*. Checks related to *Naming Conventions* are also activated. The *Headers* checks, instead, are activated in just three projects. Such checks can be useful for instance to check the presence of licensing headers. In our dataset, five projects (three also using CheckStyle) use more specific ASCATs (License and Apache-rat) for this purpose. Going more in depth, we analyzed the specific CheckStyle checks being activated. Twelve of the Checkstyle checks have been activated in twelve out of 13 projects. Some of them belong to the most activated categories: *Whitespace* (*e.g.,* FileTabCharacter, which checks that there are no tab characters in the source code), *Imports* (*e.g.,* UnusedImports, which checks that all the imported elements are used), *Coding* (*e.g.,* EqualsHashCode, which checks whether classes which override *equals* or *hashCode* actually override both of them), and *Block checks* (*e.g.,* LeftCurly, checking whether a left curly bracket should go to newline or should follow the previous statement). Five out of twelve among the most activated checks are related to *Naming conventions*. It is also worthwhile mentioning that nine projects use the *RegexpSingleLine* check, which permits the creation of customized checks by means of regular expressions. Other than for formatting purposes, we found that in two projects this has been used to disallow the usage of certain APIs, *e.g.,* `System.out.print` and

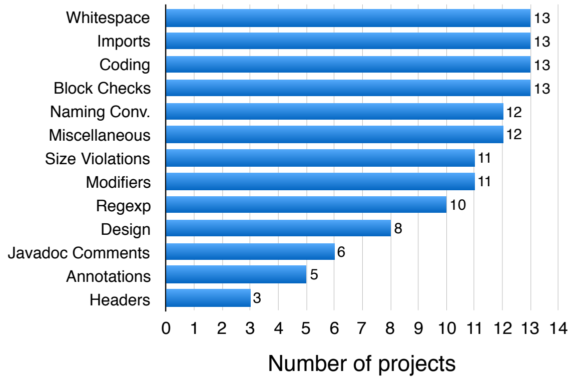

Fig. 2. Check categories activated in CheckStyle.

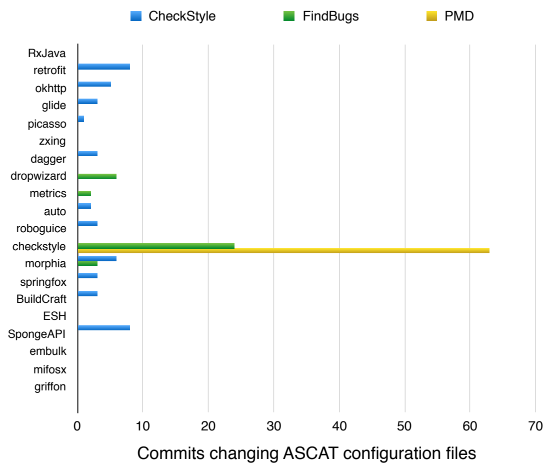

Fig. 3. Number of changes to ASCAT configurations.

`System.err.print`. In summary, compliance with coding guidelines seem to be the main reasons for using CheckStyle, which is consistent for one of the reasons why Duvall *et al.* [19] advocated the use of ASCATs in CI.

FindBugs. This tool has been customized in five projects out of 11 by suppressing some specific checks. While in four projects the suppression regarded single files or packages, in one project – *morphia* – whole categories of checks were globally suppressed. Specifically, *morphia* contains suppressions for the categories *Multithreaded correctness* (issues related to concurrency, such as inconsistent synchronization), *Internationalization* (issues related to default encoding and translations), *Security* (issues related to security, such as SQL Injection) and *Performance* (efficiency issues like explicitly invoking garbage collection).

PMD. One project out of four (*checkstyle*) includes a customization for PMD. In the analyzed history of such a project, which includes 63 different versions of the PMD configuration file, only two rule-sets where never activated, which contain Android-related and J2EE-related checks. Most of the other rule-sets, instead, resulted to be always or almost always activated (*i.e.,* disabled in < 10 versions).

2) Check suppression for some specific files: In some cases CheckStyle and FindBugs have been configured so to suppress checks on certain files. The most interesting case is related to the suppression of some checks to (JUnit) test files. This occurred in three projects using CheckStyle (*morphia*, *SpongeAPI* and *glide*), where the *JavadocMethod* check and of other checks related to the *Javadoc comments* category were suppressed. In *morphia*, also almost all FindBugs checks were suppressed to test files (except those JUnit-specific and a few others related to serialization and vulnerabilities). This confirms findings of previous works [35], [36], suggesting that developers do not perceive as particularly important the maintainability of test cases.

Other suppressions occurred, for both CheckStyle and FindBugs, in files not following specific patterns. When checks were suppressed on some files only, this happened where developer realized that either the check did not make sense, or it clearly produced false positives. For example, in *metrics* and *dropwizard* the *Transient field that isn't set by deserialization* check was deactivated for servlets where such a problem is not relevant.

3) How configurations changed over time: Fig. 3 shows the number of changes occurred to ASCAT configuration files across history. For five projects (*RxJava*, *ESH*, *embulk*, *mifosx*, and *griffon*) configuration files never changed. In other cases we found up to ten changes, with the exception of the *checkstyle* project where FindBugs and PMD had their configurations changed 24 and 63 times respectively. The latter case is particularly interesting, and we observed a number of active checks growing in time, from 14 active in the first version to 22 active in the last observed one. No rule was dropped over the history, except for security guideline checks, disabled in the last 20 versions of the configuration files. This likely indicated a lost of interest of such checks, for a program, *checkstyle*, for which security is not a big issue as for network-centric applications.

While our study agrees with findings of Beller *et al.* [13] suggesting limited changes to configuration files, such changes still occur and it is therefore worthwhile to investigate whether they influence the introduction or fix of broken or warned builds (**RQ**$_3$).

RQ$_1$ **summary.** Except in one case, ASCATs are either used to break builds or to warn them, with some tools (*e.g.,* CheckStyle) mostly used to break builds. ASCATs are almost always configured, and such configurations generally change over time a limited (< 10) number of times in the observed period. File-based suppressions occur where some checks do not make sense or, in some cases, on test code.

C. RQ$_2$: What types of issues make build warn/fail?

Table III details the number and percentage of broken and warned builds for each project and for each ASCAT the project uses. The percentage refers to the number of builds

TABLE III
FAILURES AND WARNINGS DUE TO STATIC ANALYSIS TOOLS IN CI.

System	ASCAT	# of Builds with ASCAT active/total	Broken builds for ASCAT (%)	Warned builds (%)
RxJava	FindBugs	8/391	–	3 (37.50)
	License	391/391	–	373 (95.40)
retrofit	CheckStyle	558/558	4 (0.72)	–
okhttp	CheckStyle	1,436/1,436	33 (2.30)	–
glide	CheckStyle	102/108	0	–
picasso	CheckStyle	205/205	2 (0.98)	–
zxing	CheckStyle	10/296	0	–
	FindBugs	219/296	0	–
	Apache-Rat	10/296	1 (10.00)	–
	Clirr	80/296	1 (1.25)	–
dagger	CheckStyle	250/250	1 (0.40)	–
dropwizard	FindBugs	633/633	6 (0.95)	–
metrics	FindBugs	295/295	1 (0.34)	–
auto	Checkstyle	181/181	3 (8.57)	16 (10.96)
roboguice	CheckStyle	138/200	0	–
	FindBugs	138/200	0	–
	PMD	138/200	0	–
checkstyle	FindBugs	1,455/1,486	0	–
	PMD	1,455/1,486	0	–
morphia	CheckStyle	272/272	13 (4.78)	–
	FindBugs	272/272	8 (2.94)	–
springfox	CheckStyle	288/505	4 (1.39)	–
	FindBugs	146/505	0	–
	PMD	146/505	0	–
BuildCraft	CheckStyle	1,273/1,273	444 (34.88)	–
ESH	FindBugs	624/736	–	138 (22.12)
	PMD	624/736	–	138 (22.12)
SpongeAPI	CheckStyle	880/880	–	483 (54.89)
	License	880/880	28 (3.18)	–
embulk	FindBugs	457/457	–	445 (97.37)
mifosx	License	1,756/1,756	56 (3.19)	–
griffon	CheckStyle	217/217	13 (5.99)	–
	FindBugs	95/217	–	87 (91.58)
	License	217/217	–	202 (93.09)
	jDepend	217/217	–	59 (27.19)

TABLE IV
FAILED AND WARNED BUILDS FOR EACH WARNING CATEGORY. CS: CHECKSTYLE, FB: FINDBUGS, L: LICENSE, CL: CLIRR, A-R: APACHE-RAT, JD:JDEPEND.

Category	Broken Builds			Warned Builds		
	Conf.	Failed	(%)	Conf.	Warned	(%)
Maintainability			**2.08**			**20.84**
Annotations (CS)	1,408	0	0.00	511	0	0.00
Block checks (CS)	4,784	82	1.71	1,026	36	3.51
Class design (CS)	3,758	88	2.34	559	0	0.00
Coding (CS)	4,784	168	3.51	1,026	248	24.17
Javadoc comments (CS)	1641	12	0.73	559	293	52.42
Imports (CS)	4,784	417	8.72	467	215	46.04
Metrics (CS)	288	0	0.00	–	–	–
Miscellaneous (CS)	4,279	15	0.35	1,026	430	41.91
Modifiers (CS)	4,279	30	0.70	978	5	0.51
Naming conventions (CS)	4,496	58	1.29	1,026	30	2.92
Regexp (CS)	3,314	27	0.81	146	0	0.00
Size violations (CS)	3,294	9	0.27	978	191	19.53
Whitespace (CS)	4,784	144	3.01	1,026	129	12.57
Dodgy code (FB)	2,886	1	0.03	1,089	586	53.81
Basic* (PMD)	1,739	0	0.00	624	138	22.12
Defects			**0.04**			**22.70**
Bad practice (FB)	2,886	2	0.07	1,089	488	44.81
Correctness (FB)	2,886	3	0.10	1,089	41	3.76
Experimental (FB)	2,886	0	0.00	1,089	0	0.00
Internationalization (FB)	2,886	1	0.03	1,089	92	8.45
Malicious code (FB)	2,886	0	0.00	1,089	490	45.00
Multithread correctness (FB)	2,886	0	0.00	1,089	448	41.14
Performance (FB)	2,886	3	0.10	1,089	560	51.42
Security (FB)	2,886	0	0.00	1,089	0	0.00
Basic** (PMD)	1,739	0	0.00	624	1	0.16
Removed field (Cl)	80	1	1.25	–	–	–
Field less accessible (Cl)	80	1	1.25	–	–	–
Licensing			**1.60**			**94.57**
Headers (CS)	47	0	0.00	–	–	–
Missing header (L)	2,636	84	3.19	608	575	94.57
Unknown extension (L)	2,636	0	0.00	608	575	94.57
Unapproved License (A-r)	10	1	10.00	–	–	–
Execution Issues			**0.00**			**27.19**
Unknown Constant (jD)	–	–	–	217	59	27.19

for which the ASCAT has been activated (third column). As also discussed in Section III-B, in all cases ASCATs are either configured to break the build or to raise warnings. Only the project *auto* changed the ASCAT action over time.

As Table III shows, in most cases ASCATs break a percentage of builds between 0% (see in particular *roboguice*, where no ASCAT broke any build) and 34.88% (CheckStyle in *BuildCraft*). If looking at warned builds (where ASCATs are configured to do so) percentages are generally higher than for broken builds, in many cases above 90%. This is not surprising and indicate that, when developers use a strict (*i.e.,* to break builds) ASCAT configuration, they also try to limit pushing changes with such problems, *e.g.,* by being more careful in writing the code or, possibly, by running private builds. Warnings do not interrupt the CI process, therefore high percentages can be perfectly acceptable. Moreover, as reported by Bacchelli and Bird [11] in the context of modern code review, warnings could improve modern code review tools providing a better team awareness and also transferring knowledge.

It is interesting to understand why in two cases—*checkstyle* and *roboguice*—the ASCATs never failed nor raised warnings. By analyzing the commits' messages of *checkstyle*, we found some of them explicitly mentioning the fix of PMD and FindBugs raised warnings. Moreover, in many cases the commits' messages also reported the type of rule violated and fixed, with a link that directly points to the tools' official documentation. Our explanation is that ASCATs were also run in private builds, problems fixed locally, and then no further issue was raised on the CI server. Instead, we found no such evidence for *roboguice*.

Table IV provides an analysis of broken and warned builds by category. The analysis is cumulative over builds of all projects in which each tool has been used and the checks for that category activated to either break or warn the build. The number of such builds are indicated in the second and fifth column respectively. Categories are grouped mainly following the high-level categorization suggested by Beller *et al.* [13], *i.e.,* Maintainability issues and Defect issues (for the low-level categories we preferred to keep the ones in the ASCATs' documentation). Then, we decided to create a specific category for Licensing-related issues (instead of merging it together with other Defects), for which ASCATs provide specific checks. Finally, we have a category in which we reported errors related to misuse/misconfiguration of the ASCAT.

In the following, we discuss in detail broken and warned builds related to the four categories.

Maintainability. Such issues are generally related to the adherence to coding standards/guidelines, code and design smells, and in general to choices deteriorating the source code readability and maintainability. On average, they break the build 2.08% of the times they are activated, while they warn the builds in 20.84% of the cases. The check category that induced the largest percentage (8.72%) of build breakages is the CheckStyle *Imports* one, related to checking for unused or redundant imports but also for import ordering. Also, 3.51% of the breakages were related to bad coding practices (*Coding*) and 2.34% to *Class Design* issues. Note that most of the *Imports* and *Coding* breakages (390 out of 444, and 165 over 168) are related to the *BuildCraft* project, *i.e.,* the one with the highest percentage of observed broken builds. CheckStyle also produced a relatively high number of broken builds related to coding standards, and in particular for *Whitespace* (3.01%) and *Naming Conventions* (1.29%). *Imports* checks also generated a large percentage of warnings in some other projects (46.04%), while some other checks mainly generated warned builds rather than broken builds. This is the case for *JavaDoc comments*, but also for FindBugs' *Dodgy Code*, (mostly checking for the presence of useless control flow and also the insertion of redundant null checks), for the *Size Violations* on line and method length, and for PMD *Basic*[5] maintainability checks. Given the high percentages observed, developers of projects that configured such checks to just warn the builds might not consider such problems very relevant. Instead, if one looks at the percentage of broken builds and at the number of builds in which the same checks have been activated for failure (on different projects, as the activation modes were mutually exclusive except for *auto*), realizes that, when the broken activation mode was considered, developers mostly avoided to violate the checks.

Defects. Concerning checks related to likely defects, on the one hand the overall percentage of warned builds (22.70%) is in line with the one observed for maintainability issues. On the other hand, when checks related to defects were configured to break the build (and this happened in a very large number of cases, as the second column of the Table IV shows), very few cases of broken builds occurred (on average, 0.04% of the cases). The interpretation we provide to this observation is consistent with findings of previous work indicating that, while static analysis tools have the potential to also highlight the presence of likely defects, performance problems, or vulnerabilities, in such a circumstance they also produce a high number of false positives [39]. With respect to that, our results indicate that there are either cases in which developers care about such suggestions, configure them to break the build and make sure such problems almost never occur. In other cases, developers just let the ASCAT raise warnings, without necessarily having to cope with that because of a broken build.

Licensing. Such issues were addressed by the CheckStyle *Headers* category, by the License plugin, and by Apache-rat. On average, such checks caused the breakage of 1.6% of the build in which they were activated and warned 94.57% of the builds. Such percentages are relatively high if compared with those of other check categories, and highlight on the one hand the increasing awareness about licensing issues, on the other hand typical mistakes developers commit, *e.g.,* not including licenses in headers, especially when the project has files with multiple licenses. We can also notice that, as also discussed in Section III-B, the CheckStyle *Headers* category is rarely enabled. Three of the projects using CheckStyle preferred to use more specific tools (*License* and *Apache-rat*) for that, and avoided duplicate checks. As for other projects, they preferred not to add the license check in the project. It is possible that they either decided to put licenses in a single file, or they are not concerned about licensing problems yet [38].

Execution Issues. jDepend raised warnings in 59 builds. The issue (*Unknown Constant: 18*) turned out to be a tool configuration issue (unable to properly work with Java 8) rather than a real warning raised by the ASCAT.

RQ$_2$ summary: ASCAT checks cause a fairly limited number of broken builds, mainly related to adherence to coding standards, but also some likely defects and licensing issues. Checks related to likely defects almost never make the builds fail when configured as such. When configured to warn the build – possibly because of the high number of false positives – we observe a high percentage of warnings.

D. RQ$_3$: When and how are broken/warned builds resolved?

TABLE V
BROKEN AND AND WARNED BUILDS DURATION.

Failures	**min**	**1Q**	**median**	**3Q**	**max**
			# of Builds		
Other	1	1	1	2	114
ASCAT-related	1	1	2	4	33
Warnings	1	2	5	37	456
			Time [h]		
Failures	**min**	**1Q**	**median**	**3Q**	**max**
Other	0.0002	0.4	5.0	26.3	4,336.9
ASCAT-related	0.02	0.7	7.4	38.1	1,390.2
Warnings	0.003	18.1	125.2	777.0	12,554.7

Duration of subsequent failed/warned builds. Table V reports descriptive statistics (min, max, first, second, and third quartile) of the duration of build breakages not related to ASCATs, of ASCAT-related build breakages, and of the occurrence of warnings. Durations are computed both in terms of number of subsequent failed builds and in terms of time (hours). As the table shows, the median fix time for ASCAT-related failures is seven hours and two builds, while the median fix time for other build failures is 5 hours (and a median of only one failed build). Time (125.2 hours) and number of builds (5) increases for the case of non-breaking warnings. ASCAT-related failures entail a significantly longer sequence (Wilcoxon rank sum test [17], significance level $\alpha = 0.05$) of build failures than other failures (adjusted[6] p-value<0.001, Cliff's [20] d=0.22 – small even if the median difference is 1), and as expected warnings in turn entail a significantly longer

[5]We have split the *Basic* PMD checks into those related to maintainability (labeled as *Basic**), and those related to defects (labeled as *Basic***).

[6]Using Holm's correction [22].

TABLE VI
HOW BUILD FAILURES ARE INTRODUCED.

Failure Type	**Changes to** Source Code	Build Scripts	ASCAT Config.	CI Config.	Other Files
Other	73.39%	17.81%	2.15%	4.29%	11.37%
ASCAT-related	97.83%	9.24%	3.80%	0.54%	0.00%
Warnings	90.24%	4.88%	0.00%	4.88%	4.88%

TABLE VII
HOW BUILD FAILURES ARE FIXED.

Failure Type	**Changes to** Source Code	Build Scripts	ASCAT Config.	CI Config.	Other Files
Other	72.53%	19.53%	1.72%	4.29%	10.73%
ASCAT-related	94.57%	20.65%	3.26%	1.09%	2.17%
Warnings	80.49%	12.20%	4.88%	0.00%	7.32%

sequence of builds than ASCAT-related failures (adjusted p-value=0.001, Cliff's d=0.27 – small, with a median difference of 3). Also, counting commits yields consistent results with the number of builds. Instead, if we look at duration in terms of time, there is no statistically significant difference between other failures and ASCAT-related failures (p-value=0.09, with a median difference of 2.4 hours), whereas, not surprisingly, the difference is significant between ASCAT-related failures and ASCAT-related warnings (p-value<0.001, Cliff's d=0.48 – medium, with a median difference of 117.8 hours).

Changes inducing failed/warned builds. Table VI provides a breakdown of the percentages of builds failures induced by different changes to different kinds of files (to source code, build files, ASCAT configuration files, CI configuration files, or other files). Percentages are referred to the total number of failures of each file type (note that each row may sum up above 100% because multiple types of files may have been changed when a build was broken or warned). As one can notice, unsurprisingly, in most of the cases failures are introduced due to changes to source code, and this is particularly true for ASCAT-related failures (97.83% of the cases). The second-most frequent failure-inducing change is related to build scripts. This is again not surprising because it means that new jobs (or new analyses in case of ASCAT-related failures) have been introduced. Instead, ASCAT-related failures are rarely (3.80%) introduced because of changes to ASCAT configuration files (which, by chance, are also modified in 4.29% of the cases in which other failures occur). A deeper analysis revealed one case where the build was broken as soon as ASCAT (Apache-rat) was added to the build process of *zxing*, one where some FindBugs suppressions were removed (*morphia*), and one (in *springfox*) not actually related to the ASCAT breaking the build.

Changes fixing failed/warned builds. Table VII provides a breakdown of the percentages of broken or warned builds fixed by changing different types of files. Also in this case, the role played by changes to source code is predominant for all kinds of failures, and especially for ASCAT-related build breakages (94.57% of the cases). This likely means that either the code with issues has been removed, or that it has been modified so to avoid the highlighted issues. Also, in many cases failures and warnings are fixed by changing build scripts. For example, in *ESH* there was a case in which developers disabled the ASCAT to fix the warned builds. Fixes due to changes of ASCAT configuration files occur in a fairly limited percentage of instances (3.26% for ASCAT-related build breakages and 4.88% for warnings). For example, in *springfox* project the line length threshold of the CheckStyle's *Size Violations* check was changed to avoid excessive warnings.

TABLE VIII
HOW ASCAT-RELATED BUILD FIXES ARE DOCUMENTED IN COMMIT MESSAGES.

Type	**Occ.**	**Examples**
Tool-related fix	43	Fix checkstyle violations Fix some foundbugs Fix rat failure
Warning fix	40	fixed style whitespace removal... Add missing license header
Build fix	4	Fix broken build
Tool config.	3	Disable findbugs/pmd by default (to speed up build) Convince FindBugs that ConfigurationFactory is OK Update checkstyle.xml and fix problems

Evidence on documented build fixes. Out of 238 fixes to build chains failed or warned because of ASCATs, 72 of them provide a documented evidence that developers either addressed the warnings produced by the tools, or reconfigured the build to skip the ASCAT. More specifically, we found documented evidence for 59 fixes to build failures and for 13 fixes to warned builds. Table VIII provides a classification of the kinds of commit messages we found. We manually categorized them into four categories. 43 commits explicitly mention that the fix is related to the tool causing build breakages or warnings. In 40 cases, the commit message is, instead, specifically referring the kind of problem being fixed. In many cases it relates to style issues, often raised by CheckStyle, but we also found comments related to other warnings, *e.g.,* unused code, or licensing-related fixes (to fix builds broken by the License plugin). In a few cases, the commit just refers that the failed (because of ASCATs) build has been fixed. Finally, we found three cases in which the commit especially mentioned that the tool has been either reconfigured or disabled. In one case, this was done not only to avoid build breakages, but also to speed up builds, as also suggested by Duvall *et al.* [19]. It is important to highlight that summing the occurrences in Table VIII we obtain a value greater than 72 because each commit message can belong to more than one type.

RQ$_3$ summary: ASCAT-related broken builds are almost immediately fixed (in median within 8 hours), in most cases by actually fixing the problem in the source code, and only in few cases by modifying the build script or the ASCAT configuration. In 30% of the cases, commit messages document the activity performed to solve ASCAT-related issues.

IV. Threats to validity

Threats to *construct validity* concern the relationship between theory and observations. Results of $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$ are based on the analysis of build logs and ASCAT outputs using regular expressions, and on their classification into high-level categories. While it has been always possible to unambiguously define a regular expression to match a warning type, and while for the high-level categorization we relied on the tool documentation, it is still possible that mistakes could have introduced imprecisions in our results.

The analysis performed in $\mathbf{RQ}_3$ on the duration of build failures only provide a proxy of the difficulty in fixing a problem or of the effort developers put in it. Indeed, the reasons for trying several consecutive builds before fixing a problem or of taking much time before a build success can be due to other reasons that are not easy to observe.

Threats to *internal validity* concern factors internal to our study that could influence our results. The analysis of build types performed in $\mathbf{RQ}_2$ and reported in Table IV relies on project rebuilds performed locally. Although we replicated the configuration found on the actual project builds, there is still the possibility that our local builds produced after some time may have produced different results. At minimum, we checked that the overall outcome of the build phase related to ASCAT was the same observed on Travis CI. As we said, we had to exclude FindBugs results for two projects: *griffon* and *morphia*. Moreover, we could have omitted the analysis of rebased commits (*e.g.,* after a pull request review) for which ASCATs would have raised warnings. However, when rerunning the build locally, we verified at least the consistency of build failure types produced by ASCATs with the information available in the Travis CI logs.

Threats to *external validity* concern the generalization of our findings. This study is intentionally limited to 20 Java projects, because we preferred at this stage an in-depth evaluation of the analyzed projects from various perspectives. Also, as explained in Section II-B the analyses of $\mathbf{RQ}_2$ required, in some cases, to locally reproduce each broken or warned build. Clearly, the study needs to be complemented with further ones, above all extending the analysis to other programming languages and, possibly, observe the use of ASCATs in industrial projects.

V. Related work

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

A. Continuous Integration and Build Failures

In recent years, researchers have studied the CI practices adopted in industry and open source. As reported by Ståhl *et al.* [34], CI is becoming increasing popular in software development. This is confirmed by Hilton *et al.* [21], who conducted an extensively study about the usage of CI infrastructure in open source projects showing how this practice has become increasingly popular in OSS. Instead, as regards the adoption of CI in industry, Laukkanen *et al.* [27] noticed how developers, adopting CI process, deal with several technical and social challenges like the test infrastructure. Moreover, Ståhl *et al.* [33] showed that in industry there is not a unique and homogeneous CI practice. Vassallo *et al.* [37] investigated the adoption of CI in a large financial organization by surveying 158 DevOps. Among other results, they indicated how ASCATs within CI are mainly used to get rid of unused/unreachable code. This is consistent with some of the checks we found highly used like CheckStyle's *Imports*.

Previous researchers have also investigated on the nature and impact of build failures (in CI and not). Miller [28] reported on a study conducted at Microsoft and showed that builds mainly fail for static analysis (40%), unit testing, compilation and server failures. In our study, the observed percentage of broken builds due to ASCATs is much smaller than what observed by Miller. Differently than Miller, we perform a deep investigation on the nature of build failures due to ASCATs, other than looking at their percentage and their removal.

Studies conducted by Seo *et al.* [31] and Beller *et al.* [14] focused on compilation and testing build failures, respectively. Kerzazi *et al.* [24] measured the impact of build failures in a software company, analyzing more than 3k builds, and also tried to identify possible causes and effects interviewing 28 software engineers.

B. Usage of static analysis tools

Different researchers observed how ASCATs are used during software development. Kim and Ernst [26] analyzed the warnings identified by three different ASCATs, showing that only a small percentage of them ($\simeq$ 10%) is removed during fix-changes' operations. Spacco *et al.* [32] tracked warnings/errors raised by FindBugs to projects' defects, and measured the lifetime of specific warnings/errors categories. In our study, we show how the use of ASCATs in the CI pipeline is a way to force a quick and consistent removal of some warnings. Couto *et al.* [18] analyzed the actual gains in using ASCATs during the development process showing that there is not a static relationship between warnings and field defects even if there is a weak level of correlation between them.

Wedyan *et al.* [39] showed that the usage of ASCATs is more effective to suggest refactoring opportunities than for detecting faults. Moreover, they showed that ASCATs usually produce a high number of false positives ($\simeq$ 95%). Our results confirm the usage of ASCATs mainly for refactoring and code standard adherence, less for potential bugs probably related to the high number of false positives. Zheng *et al.* [41], highlighted the important role played by ASCATs in the context of software fault detection and also that there is a high percentage of warnings mainly related to programmer's errors. Ayewah *et al.* [10] conducted a study at Google, showing that $\simeq$ 70% of warnings/errors produced by FindBugs are real defects, even if, quite surprisingly, only few of them causes noticeable problems in production. Moreover, many of them strictly correlate to problems/issues in which reviewers are interested in. Johnson *et al.* [23] interviewed 20 developers to investigate the reasons for the under-usage of ASCATs. The study revealed that the large number of false positives and the inadequate understandability of the output generated by the tools are reasons for such under-usage, especially because tools do not produce adequate information to simplify the process for finding and fixing a bug. Finally, Rahman *et al.* [30] compared static analysis tools and statistical defect prediction tools, showing that in several cases ASCATs and statistical methods produce comparable results, although there are cases in which tools like FindBugs work better than statistical methods. Since the above studies suggest that only some warnings turn out to be true positives and relevant for developers, a careful configuration of ASCATs is required, especially within, CI as we noticed.

Panichella *et al.* [29] investigated the usage of ASCATs during code reviews, measuring the density of warnings after each review and showing that this copes with a small variation. Moreover, when they focused on specific kind of warnings, they were able to identify categories for which the reduction in the measured density of warnings after a specific review is higher than the previous case (from 50% up to 100%).

Recently, Beller *et al.* [13] conducted a large scale evaluation on the usage of ASCATs in OSS. Their results underline that ASCATs usage is not widespread in popular software systems and also that their usage is strictly related to the programming language of the analyzed projects. Even if Beller *et al.* did not observed the use of ASCATs in the context of CI, they pointed out the increased benefits of ASCATs when integrated in the project development workflow (and CI process in particular). This observation motivates our work. Moreover, as regards the configurations mainly used by the developers, Beller *et al.* showed that there is $\simeq 5\%$ in which developers do not rely on default configurations of ASCATs, and therefore reconfigure them. Moreover, they found that, even though ASCATs tend to be reconfigured, this often happens once in the project lifetime, and reconfigurations seldom occur. While our results generally agree with such findings, we still found a number of changes, and in some cases (*e.g.,* PMD configurations for the *checkstyle* project), a conspicuous number of changes. Finally, Beller *et al.* showed that only in few cases (10%) developers tend to document issues/fixes related to ASCATs warnings, while we highlight a good percentage of documented fixes (30.25%) when those are performed to solve CI build breakages.

VI. Lessons Learned and Conclusions

This paper reported a study aimed at investigating the practices of using Automated Static Code Analysis Tools (ASCATs) within Continuous Integration (CI) in 20 popular Java open source projects hosted on GitHub and using Travis-CI as CI infrastructure. We investigated which ASCATs such projects use, how they are configured in CI, what build failures or warned builds they generate, when and how such failures and warnings are resolved. Differently from the use in private builds, the use of ASCATs in CI produces a "stronger" message in case the tool reveal warnings, letting everybody be aware about the issue and, in some cases, producing build breakages. At the same time, however, such an effect suggests an use of ASCATs with parsimony to avoid many unnecessary build breakages.

Results of the study provide some suggestions for a more effective and efficient adoption of CI (in parentheses we refer to the research question which results determine the suggestion):

- When introducing ASCATs in the CI process, developers should understand what checks do not make sense for their project (*e.g.,* web-specific checks if your project is not a web application) and exclude them to avoid immediate, repeated and obvious failures (from $\mathbf{RQ}_1$), but also to avoid slowing down the CI process (from $\mathbf{RQ}_3$).
- One may decide to exclude non-production code (*e.g.,* test) from static analysis, but keep in mind it may exhibit maintainability and understandability problems too (from $\mathbf{RQ}_1$).
- Forcing adherence to coding guidelines is one of the most useful applications of ASCATs in CI; thus, the tool needs to be properly configured according to internal coding guidelines (from $\mathbf{RQ}_{1-2}$).
- Likely defects, performance, concurrency and security checks can be useful, however they may produce many false positives (as previous literature suggested [39]), which could introduce unnecessary build breakages. Thus, such checks should rather be used to warn builds only (from $\mathbf{RQ}_2$).
- Avoid duplicate checks. This confirms what also advocated by Duvall *et al.* [19], p. 95. For example, our results indicated for licensing check the preference of License and Apache-rat over CheckStyle, and for coding standards CheckStyle (more customizable) over FindBugs (from $\mathbf{RQ}_1$).
- ASCATs within CI can provide effective facilities to help avoiding some simple licensing problems (checking missing licenses or for wrong licenses). Unfortunately, they do not suffice for more advanced analyses, *e.g.,* checking for licensing compatibilities, more sophisticated tools should be included in CI (from $\mathbf{RQ}_{1-2}$).
- Make specific commits to suppress warnings, documenting the decision rationale, so that in future others can learn from it (from $\mathbf{RQ}_3$).

Clearly, the obtained findings and lessons are based on data from the 20 Java open source projects we analyzed. Therefore, it is desirable to complement the study with further analysis on a larger dataset on a larger dataset and on other programming languages, but also with specific surveys directly involving developers from open source and from industry. Also, similarly to what it was done in the past to prioritize the fix of static warnings in a more general context [26], we plan to mine the collected data from ASCATs usage to automatically provide recommendations on prioritizing checks within CIs.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[21] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 426–437.

[22] S. Holm, "A simple sequentially rejective Bonferroni test procedure," *Scandinavian Journal on Statistics*, vol. 6, pp. 65–70, 1979.

[23] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[24] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," in *30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 41–50.

[25] S. Kim, S. Park, J. Yun, and Y. Lee, "Automated continuous integration of component-based software: An industrial experience," in *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE, 2008, pp. 423–426.

[26] S. Kim and M. D. Ernst, "Which warnings should I fix first?" in *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2007, pp. 45–54.

[27] E. I. Laukkanen, M. Paasivaara, and T. Arvonen, "Stakeholder perceptions of the adoption of continuous integration - A case study," in *Agile Conference (AGILE)*, 2015, pp. 11–20.

[28] A. Miller, "A hundred days of continuous integration," in *Agile 2008 Conference*, 2008, pp. 289–293.

[29] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?" in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 161–170.

[30] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu, "Comparing static bug finders and statistical prediction," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 424–434.

[31] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at Google)," in *Proc. Int'l Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 724–734. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568255

[32] J. Spacco, D. Hovemeyer, and W. Pugh, "Tracking defect warnings across versions," in *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 2006, pp. 133–136.

[33] D. Ståhl and J. Bosch, "Automated software integration flows in industry: A multiple-case study," in *Companion Proc. Int'l Conf. on Software Engineering (ICSE Companion)*, 2014, pp. 54–63.

[34] ——, "Modeling continuous integration practice differences in industry software development," *J. Syst. Softw.*, vol. 87, pp. 48–59, Jan. 2014.

[35] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, 2016, pp. 4–15.

[36] A. van Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*, 2001, pp. 92–95.

[37] C. Vassallo, F. Zampetti, D. Romano, M. Beller, A. Panichella, M. Di Penta, and A. Zaidman, "Continuous delivery practices in a large financial organization," in *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 41–50.

[38] C. Vendome, M. L. Vásquez, G. Bavota, M. Di Penta, D. M. Germán, and D. Poshyvanyk, "License usage and changes: a large-scale study of java projects on github," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015*, 2015, pp. 218–228.

[39] F. Wedyan, D. Alrmuny, and J. M. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction," in *2009 International Conference on Software Testing Verification and Validation*. IEEE, 2009, pp. 141–150.

[40] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta, "How open source projects use static code analysis tools in continuous integration pipelines. Replication Package http://home.ing.unisannio.it/fiorella.zampetti/datasets/MSR_2017_ASCATsCI.zip."

[41] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE transactions on software engineering*, vol. 32, no. 4, pp. 240–253, 2006.