How Developers Engage with Static Analysis Tools in Different Contexts

Vassallo, Carmine; Panichella, Sebastiano; Palomba, Fabio; Proksch, S.; Zaidman, A.E.; Gall, HC

DOI
10.1007/s10664-019-09750-5

Publication date
2020

Document Version
Final published version

Published in
Empirical Software Engineering

Citation (APA)
Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A. E., & Gall, HC. (2020). How Developers Engage with Static Analysis Tools in Different Contexts. *Empirical Software Engineering*, *25*(2), 1419-1457. https://doi.org/10.1007/s10664-019-09750-5

Empirical Software Engineering (2020) 25:1419–1457
https://doi.org/10.1007/s10664-019-09750-5

How developers engage with static analysis tools in different contexts

Carmine Vassallo[1] · **Sebastiano Panichella**[2] · **Fabio Palomba**[1] · **Sebastian Proksch**[1] · **Harald C. Gall**[1] · **Andy Zaidman**[3]

Published online: 25 November 2019

Abstract

Automatic static analysis tools (ASATs) are instruments that support code quality assessment by automatically detecting defects and design issues. Despite their popularity, they are characterized by (i) a high false positive rate and (ii) the low comprehensibility of the generated warnings. However, no prior studies have investigated the *usage* of ASATs in different *development contexts* (e.g., code reviews, regular development), nor how open source projects integrate ASATs into their workflows. These perspectives are paramount to improve the prioritization of the identified warnings. To shed light on the actual ASATs usage practices, in this paper we first survey 56 developers (66% from industry and 34% from open source projects) and interview 11 industrial experts leveraging ASATs in their workflow with the aim of understanding how they use ASATs in different contexts. Furthermore, to investigate how ASATs are being used in the workflows of open source projects, we manually inspect the contribution guidelines of 176 open-source systems and extract the ASATs' configuration and build files from their corresponding GitHub repositories. Our study highlights that (i) 71% of developers do pay attention to different warning categories depending on the development context; (ii) 63% of our respondents rely on specific factors (e.g., team policies and composition) when prioritizing warnings to fix during their programming; and (iii) 66% of the projects *define* how to use specific ASATs, but only 37% *enforce* their usage for new contributions. The perceived relevance of ASATs varies between different projects and domains, which is a sign that ASATs use is still not a common practice. In conclusion, this study confirms previous findings on the unwillingness of developers to configure ASATs and it emphasizes the necessity to improve existing strategies for the selection and prioritization of ASATs warnings that are shown to developers.

Keywords Static analysis tools · Development context · Continuous integration · Code review · Empirical study

Communicated by: Massimiliano Di Penta

This article belongs to the Topical Collection: *Software Analysis, Evolution and Reengineering*
Guest Editors: Massimiliano Di Penta and David Shepherd

✉ Carmine Vassallo
vassallo@ifi.uzh.ch

Extended author information available on the last page of the article.

1 Introduction

The increasing complexity of modern software systems has complicated both the development of new software features and the maintenance of source code (Lehman 1980). This is especially true when considering the difficulties of developers to find defects or design issues in changes to the source code (Catolino et al. 2018; Palomba et al. 2017; Parnas and Lawford 2003). Manual processes like code review (Bacchelli and Bird 2013) exist to (i) ensure the quality of source code, (ii) verify the correctness of bug fixes (McIntosh et al. 2014; Rigby and German 2006), (iii) enforce coding conventions (Beller et al. 2014), or (iv) improve maintainability (Balachandran 2013; Rigby 2011). However, the *manual effort* of code reviews is considerable (Jørgensen 2004) and defect detection is a *very error-prone* activity (Chen 2015; Khoshgoftaar and Allen 1998).

Automatic Static Analysis Tools (ASATs), i.e., tools that analyze code quality characteristics without program execution, represent an excellent opportunity to make this activity more efficient. Several tools exist (e.g., CHECKSTYLE 2019, PMD 2019) that can support developers in various tasks like the detection of defects Di Penta et al. 2009; Hovemeyer and Pugh 2004; Coverity 2009), design issues (Beller et al. 2014), code style violations (Johnson 1977), or to perform formal verification (D'silva et al. 2008). Previous research has shown that ASATs can help in detecting software defects faster and cheaper than human inspection or testing would (Beller et al. 2015a, b, 2017; Johnson et al. 2013). As such, ASATs are regularly integrated in contemporary open source (Beller et al. 2016) and industrial (Sadowski et al. 2015; Vassallo et al. 2016; Emanuelsson and Nilsson 2008) projects.

The advantages of ASATs are overshadowed by (i) high false-positive rates, i.e., alerts that are not actual issues, (ii) a low understandability of the alerts, and (iii) a lack of automated quick fixes for identified issues (Johnson et al. 2013). As a result, previous work found that only 10% of the suggested warnings of typical ASATs are actually removed during bug fixing activities (Kim and Ernst 2007). To improve this number, it is not only required to improve the precision of ASATs, it is also crucial to make it easier for the developer to spot the relevant warnings, for example, through better prioritization strategies (Beller et al. 2016). However, ASATs are being used in different *development contexts* and previous results suggest that developers use ASATs differently in these contexts. For example, Panichella et al. (2015) found coding-structure related warnings to be the most frequently fixed category in *code reviews*, while Zampetti et al. (2017) found that ASAT-related build failures are mainly caused by coding standard violations.

In this paper, which is an extension of our previous work (Vassallo et al. 2018), we analyze *where* developers use ASATs and *how* they use ASATs in these contexts. We address three main research questions:

RQ$_1$ In which development contexts do developers use ASATs?
RQ$_2$ How do developers configure ASATs in different development contexts?
RQ$_3$ Do developers pay attention to the same warnings in different development contexts?

Through a survey study involving 56 developers[1] (66% working in the industry and 34% open source contributors) and semi-structured interviews with 11 industrial developers, we

[1]Compared to our previous work (Vassallo et al. 2018), we collected 14 more participants

obtain two key findings. We validate that the prevalent development contexts for ASAT use are *continuous integration*, *code review*, and *local programming*. In addition, our participants state that they use the same ASAT configuration in these contexts, but that, depending on the context, they pay attention to a different set of warnings. We conclude that more effective use of ASATs could leverage information about the development context for a better selection and prioritization of ASAT warnings.

In this extension, we build upon the initial results on *how* ASATs are being used and analyze the way open-source projects define them and enforce their use. Specifically, we study (i) whether the adoption of ASATs is **relevant** or considered mandatory for contributing to a project (e.g., pull request must not introduce warnings) and (ii) if specific types of checks (or configurations) of ASATs are **enforced**. We also investigate the general perception of the ASAT's relevance for developers. We ask three additional research questions:

RQ$_4$ Do open-source projects define ASATs usage[2] in their repository?
RQ$_5$ Is a ASATs usage[2] enforced for contributions to open-source projects?
RQ$_6$ What is the developer's perspective on the relevance of ASATs?

To address these questions, we conduct a mixed-methods research approach with both quantitative and qualitative analyses (Johnson and Onwuegbuzie 2004). First, we manually analyze the contribution guidelines and ASAT configuration files of 176 open-source projects hosted on GitHub to understand how ASATs are defined and whether their usage is enforced for new contributions. Then, we create posts on a discussion website (Reddit) to collect diverse opinions on the relevance of ASATs in practice. Our study shows that 66% of the investigated projects define how ASATs should be used for contributions, but that only half of them (37%) enforce their usage for new contributions, which shows that the ASAT usage is still limited in practice. The online discussions reveal that many developers recognize the potential of ASATs, but also that ASATs are not ready to be used regularly. It seems that a higher precision and more advanced selection and prioritization strategies are needed to enhance the developers' confidence in such tools and spread their usage in practice.

In summary, this paper provides the following contributions:

1. We explore the practical use of ASATs in a survey with 56 participants;
2. We conducted semi-structured interviews with 11 participants to validate our findings from the survey;
3. We are the first to show the potential value of considering the *development context* in ASATs;
4. We discuss insights of a manual inspection of ASAT-related contribution guidelines and resources of 176 open-source projects;
5. We present the results of discussions triggered on five forum groups related to software development;
6. We provide insights and potential implications for both ASAT vendors and researchers interested in improving techniques for the automated configuration and prioritization of warnings.

[2]In the rest of the paper, we omit the word "usage" while referring to the definition and enforcement of ASATs usage for the sake of better readability.

2 Overview of the Research Methodology

Originating from the agile coding movement, it is reasonable to believe that modern software development processes are typically structured around three well-established contexts, i.e., *local programming* (LP), *continuous integration* (CI), and *code review* (CR).

Local programming takes place in the IDEs and text editors in which developers write code. ASATs are typically added to those environments in the form of plugins and point developers to immediate problems of the written source code, like coding style violations, potential bugs in the data flow, or dead code. Developers change perspective in *code reviews* when they inspect source code written by others to improve its quality. This task is often supported through defect checklists, coding standards, and by analyzing warnings raised by ASATs (Panichella et al. 2015). The typical workflow in *continuous integration* is different: committed source code is automatically compiled, tested, and analyzed (Beller et al. 2017a; Hilton et al. 2016). ASATs are typically used in the analysis stage to assess whether the new software version follows predefined quality standards (Zampetti et al. 2017).

In this paper, we conjecture that the described *development contexts* play an important role in the adoption and configuration of ASATs and in the way actionable warnings are selected. Moreover, ASATs are very well known tools, but we conjecture that their enforcement might be notably influenced by several factors.

Figure 1 shows an overview of our methodology that we have used to test our conjectures. We started exploring the contexts where developers use ASATs and how they configure ASATs in such contexts through a questionnaire ①. Then we extended the questionnaire and conducted semi-structured interviews to analyze the impact of development contexts on the ASATs configuration ②. Finally, we conducted a quantitative analysis of the relevance of ASATs in open source projects ③ that we complemented with a qualitative analysis of this phenomenon using discussion groups ④.

2.1 RQ_{1-2}: the Development Contexts Integrating ASATs

To analyze the contexts in which developers use ASATs (RQ_1) and how developers configure them in the various contexts (RQ_2) we designed a questionnaire, implemented using *Google Forms*[3] and publicly available in our online appendix (Vassallo et al. 2019).

As a first step, we advertised the study on social media channels to acquire study participants. Then, to address more participants, we also applied *opportunistic sampling* (Gibbs et al. 2007) to find open source contributors (OSS) that adopt ASATs in their development process. We have identified matching OSS projects from the TravisTorrent dataset (Beller et al. 2017b) by searching for ASAT-related configurations in their repositories. To avoid sending unsolicited mass emails, we only asked a random sample of 52 developers of these projects for their participation.

The survey was available for three months —from June 2017 to August 2017— in order to collect as many replies as possible. However, over the course of this work, we realized that additional questions were required to answer a new research question (see Section 2.2), so we extended the initial set of survey questions. The second survey, which kept the original questions untouched, was originally open from September 2017 to October 2017 and then, from August 2018 to October 2018. We announced the extended version of the survey over the same social media channels and posted the survey on Reddit (2019) in the *Javascript*

[3] https://gsuite.google.com/products/forms/

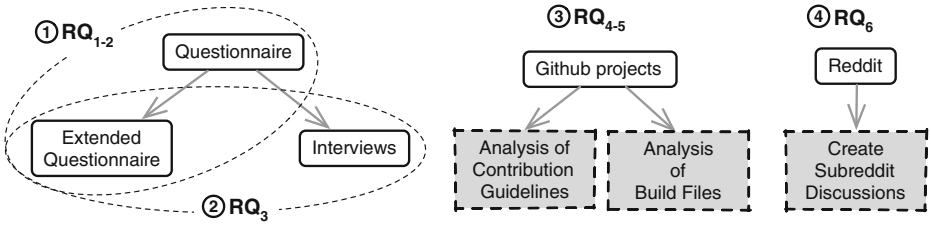


Fig. 1 The Four Steps of the Research Methodology

and *Python* communities. These communities have been selected as they (i) allow users to post surveys (unlike other suitable communities, such as *Java*) and (ii) have a large number of active subscribers, thus increasing our potential audience (e.g., the *Javascript* community has approximatively 300 daily users). In total, we received 58 responses (19 from the first survey and 39 from the second one), but we had to discard 2 of them because the corresponding respondents declared that they do not use ASATs and were, therefore, not able to properly answer our questions.

Table 1 lists demographic information about our survey participants. We had 37 (66%) industrial and 19 (34%) open-source developers. Our participants have a very diverse background. A dominant group does neither exist when split by team size, nor when split by project size. Most of our participants are experienced developers. When asked for a self-estimation of their own development experience, most of them would rate themselves as "very good" (51%) or "excellent" (36%) developers. Furthermore, 77% of them have more than 5 years of development experience, and 41% even more than 10.

We were also interested in profiling the tools our participants use during development. MAVEN (2019) (33%) and GRADLE (2019) (23%) are the (CI) build tools most commonly used by our participants. However, some participants rely on build tools like SBT (2019) (4%), that is mostly used in Scala development, or BUNDLER (2019) (2%), the most com-

Table 1 Demographic information about our survey participants

Team Size		Projects Size [LoC]	
1–5	35%	1,000-300,000	80%
5–10	31%	300,000-1,000,000	16%
10–15	14%	>1,000,000	4%
>15	20%		
Experience (Years)		Experience (Rate)	
1–5	23%	Poor	0%
5–10	36%	Fair	0%
>10	41%	Good	13%
		Very Good	51%
		Excellent	36%

mon build tool for Ruby. Only 2% of participants combine command line scripts to build the project.

Pull requests form a well-known method for collaborating and sharing opinions (Gousios et al. 2014, 2015). The largest part of our respondents declared to be supported by distributed version control systems such as GitHub (2019) (29%), Gitlab (2019) (18%) or Bitbucket (2019) (9%) during the code review process. Nevertheless, some participants still tend to rely on a dedicated code review tool, i.e., Gerrit (2019) (18%), or to use an informal process (15%).

2.1.1 ASAT Types

While answering RQ_1 we investigated which ASATs were most often used. Later (in RQ_4 and RQ_5) we also analyzed which ASATs are most frequently defined and enforced. To gain further insights useful for our analyses we have grouped all the resulting tools according to their types in the existing taxonomy of Novak et al. (2010). This taxonomy uses several dimensions like *number of releases per year*, *supported languages*, *configurability* to categorize ASATs. Since the taxonomy dates back to 2010 and the list of categories is outdated in some cases (e.g., Findbugs 2019 is categorized as *General* and *Style*, while it is well-known for spotting bugs Ayewah et al. 2008), we decided to adapt the original categories for our mapping. More specifically, we (i) removed the "General" category, because its description is too vague, (ii) merged the "Buffer Overflow" and "Security" categories, as the former represents a specific instance of the latter, and (iii) added a new category called "Correctness", which includes ASATs that search for misused methods and types. The final set of categories is illustrated in Table 2.

For our analyses, we grouped ASATs according to their provided functionalities (i.e., the *rules* dimension in the taxonomy). Two authors mapped the ASATs that were indicated in our preliminary survey in Section 3 and were defined or enforced in open-source projects as described in Section 5 to the *rules* categories. This mapping was performed in two iterations: First, one author mapped each ASAT to one or more categories. Second, a second author verified the adaptation of the original taxonomy, agreed that no further categories are needed, and mapped all ASATs to the categories as well. The mappings of both authors matched perfectly, which eliminated the need for further iterations, and are available in our online appendix (Vassallo et al. 2019).

Table 2 Taxonomy of ASATs (derived from Novak et al. 2010)

Type	Description
Style	Inspect the visualization look of the source code
Naming	Review if the variables are correctly named (e.g., naming standards)
Concurrency	Errors with concurrency running code
Exceptions	Errors by throwing or not throwing exceptions
Performance	Errors with performance of the application
Security	Errors which could impact security of the application
SQL	Searches for "SQL injections" and other SQL errors
Maintainability	Rules for better maintainability of the application
Correctness	Methods and types correctly used (according to their purpose) (e.g., Method may return null, but is declared @Nonnull)

2.2 RQ_3: The Impact of Development Contexts on the Configuration of ASATs

To investigate how development contexts influence the selection of warnings to which developers react (RQ_3), we extended our previous questionnaire (as described in Section 2.1) to include questions about the way the usage of ASATs is perceived in such contexts. We also interviewed industrial experts that use ASATs on a daily basis. The interviews complemented the extended questionnaire, as they provided another perspective on its results and could possibly explain observations coming from it.

We defined a guideline for the interviews but decided to adopt a semi-structured interview format (Runeson and Höst 2009) that allows the interviewees to guide the discussion, which possibly leads to unexplored areas. We were prepared to conduct the interviews both in person or remotely (using Skype) depending on the preference of the participant. While we took notes in the personal interviews, each remote interview has been recorded and transcribed. Through reaching out to personal contacts, we found 11 professional developers for our interviews. Our interviewees work in 6 different companies and, as shown in Table 3, they cover different domains. Specifically, 4 of them are classic software engineers, while the other 7 lead the development team where they are working or design the overall architecture of a project. Thus we had participants from both perspectives: (i) developers that actually use ASATs and (ii) developers that have to "negotiate" the expected product quality with the stakeholders and configure their ASATs accordingly. Moreover, all of them use ASATs during several activities. The majority (82%) include ASATs in their CI build. A popular choice among our interviewees is SonarQube (2019) (40%), a result that is in line with previous work conducted in the industry (Vassallo et al. 2017). The other ASATs that are most-employed in our participants' companies are FindBugs (2019) (13.6%), Checkstyle (2019) (9.1%) and IDE plugins, e.g., CodePro (2019) (9.1%).

2.3 RQ_{4-5}: the Relevance of ASATs in Open-Source Projects

We quantitatively studied the definition (RQ_4) and enforcement (RQ_5) of ASATs in open-source projects by mining project-related information on GitHub and by manually

Table 3 Demographic information about interviewees

		Organization		
Subject	Years	Role	Domain	Size
S1	20	Software Engineer	IT consultancy	100,000
S2	8	Team Lead	Financial Services	800
S3	35	Software Architect	IT consultancy	5,000
S4	8	Product Owner	Financial Services	800
S5	10	Team Lead	Financial Services	800
S6	8	Solution and Technical Architect	Financial Services	800
S7	26	Team Lead	Content Management	100
S8	11	Technology Team Lead	Financial Services	800
S9	10	Software Engineer	Services and Innovation	70,000
S10	7	Software Engineer	Financial Services	100
S11	12	Software Engineer	Financial Services	70

analyzing contribution guidelines. We wanted to observe how ASATs usage is influenced by the projects' culture and thus, by the enforced *contribution guidelines*. In this way, we could measure the relevance of ASATs in open-source projects.

We started our analysis by sampling the top-rated projects (more details in Section 5), related to the main programming languages —Java, Javascript, Ruby, and Python— that emerged in the first study (see Section 3). For each language, we selected the 50 most popular projects on GitHub (2019) (based on the number of stars) and created an initial set of 200 projects. Through reviewing the project descriptions, we discarded 24 candidates that were not software projects, but collection of the books or code snippets used as support for learning courses. We ended up using a final set of 176 projects, for which we manually analyzed (i) the ASATs' configuration files available in the projects' repositories, (ii) their build configuration file, and (iii) the project's documentation available in the repository (e.g., `README.md` files, and contribution guidelines, e.g., `CONTRIBUTING.md` files) to gather information about the actual relevance of ASATs in practice (more details about our inspection procedure in Section 5).

It is important to mention that, differently from previous work by Beller et al. (2016), we do not only measure the popularity of ASATs, but we also investigated the types of warnings for which ASATs are usually enforced.

2.3.1 Project Types

As previously described, we analyzed open source projects available on GitHub (2019. Besides categorizing projects by language, we decided to further categorize them according to their *age*, *contribution*, and *popularity* levels to gain more insights into the relevance of ASATs in open-source. We used the GitHub API (Github 2019) to request (i) the number of performed commits (to measure the *age*), (ii) the number of contributors (for *contribution* metric), and (iii) the numbers of stars (to measure the *popularity*) of a certain project. For each considered perspective (i.e., *age*, *contribution*, and *popularity*) we split projects into three different subsets, i.e., *low*, *medium*, and *high*. Specifically, we calculated the first (Q_1) and the third (Q_3) quartile of the distribution representing the number of commits, contributors, and stars of the subject systems. Then, we classified them into the following levels: (i) *low*, if they have a number of commits/contributors/stars n lower than Q_1, (ii) *medium*, if $Q_1 \leq n < Q_3$, and (iii) *high*, if n is higher than Q_3. The number of projects belonging to each level is reported in Table 8.

2.4 RQ_6: The Developers' Perspective on the Relevance of ASATs

We created discussion groups on Reddit (2019) to investigate the developers' perspective on the use of ASATs and gain qualitative insights on their relevance in practice (RQ_6). Based on the results of our investigation in the open-source community (see Sections 5.2 and 5.3), we asked our participants to reflect on their ASAT use and its importance as part of the development process. We created a post in five popular Reddit communities to gather as many replies as possible. We selected the `learn-programming` community because the community is focused on teaching how to properly develop code. Other communities have been selected based on the investigated programming languages (i.e., `Java`, `Javascript`, `Python`, and `Ruby`). We wanted to acquire feedback from developers that are used to discussing their programming and software engineering approaches. We had first considered acquiring this feedback in a survey, but such a survey would have attracted a more general selection of developers. We concluded that Reddit is the bet-

ter option because it allows targeting specific communities, with developers that are more likely to have the specific expertise required for our qualitative investigation, i.e., experience with the ASATs described and discussed in Section 3. Links to our posts in the respective communities are available in the replication package (Vassallo et al. 2019).

In total, we monitored the posts for one week and received 37 comments from 29 different subscribers. We had to discard 8 out-of-scope comments and ended up with a total of 29 comments for analysis (45% of the comments are from `java`, 28% from `python`, 17% from `javascript`, and 14% from `ruby` communities). The discarded comments are all from the `learn-programming` community. The comments that we received in such a subreddit only refer to the relevance of ASATs usage as a topic for that community. Finally, we performed open card sorting (Spencer 2009) of the comments to elicit the main statements of the discussions.

3 The Development Contexts Integrating ASATs

The *goal* of this preliminary study (as explained in Section 2.1) is to understand (i) what the development contexts are in which developers adopt ASATs (RQ_1) and (ii) how developers configure them in the various contexts (RQ_2), by surveying people that use ASATs either in open source or industrial projects. Hence, the *context* of our study includes (i) as *subjects* the participants to our survey (see Table 1) and (ii) as *objects*, the specific ASATs used by our respondents.

3.1 Survey Design

Our initial questionnaire (see "Questionnaire" in Fig. 1) consisted of 19 questions, which include 8 multiple choice (MC), 4 checkboxes (C) and 7 open (O) questions. Furthermore, we asked our participants to rate the validity of 4 statements (S) and also provided them with an opportunity to leave further comments. In Table 4 we have grouped our various questions into three topics: (i) Background, (ii) Adoption (of ASATs), and (iii) Configuration (of ASATs).

The BACKGROUND questions provided us with the demographic information that we have reported in Section 2. However, for brevity, we omit these questions in the table.

The questions in the other two sections, ADOPTION and CONFIGURATION, present the core part of the survey and aim at understanding ASATs usage in practice. Specifically, the ADOPTION section was aimed at assessing the degree of integration of ASATs in the daily development. To reach this goal, we initially asked participants how frequently they use ASATs (Q1.1), verifying whether there were some of them that never use static analysis tools during their activities. Then, we surveyed our respondents about the development activities where they usually rely on ASATs (Q1.2), specifying the mostly used types of ASATs (e.g., PMD, Findbugs, etc.) (Q1.3). Furthermore, we wanted to understand whether they use multiple ASATs (Q1.4) and in which development contexts (Q1.5). In the CONFIGURATION section (Q2.1-Q2.7) we have focused on confirming/rejecting previous results reporting how developers usually avoid the modification of the ASATs default configuration (e.g., the ones reported by Beller et al. 2016). For this reason, we asked our participants when and which are the contexts where they change the configuration of ASATs. Then we asked our respondents how frequently they fix warnings suggested by ASATs in the different considered contexts.

Table 4 Survey Questions (MC: Multiple Choice, C: Checkboxes, O: Open answer, #: the number of respondents answering the corresponding question)

Section	Summarized Question	Type	#
Adoption			
Q1.1	To what extent do you use ASATs during your activities?	MC	56
Q1.2	During which activities do you use ASATs?	O	48
Q1.3	Which ASATs do you usually work with?	C	55
Q1.4	If you use more than one ASAT, why you're adopting more than one ASAT and in which context?	O	31
Q1.5	In which step of software development do you usually rely on the suggestions provided by ASATs?	C	55
Configuration			
Q2.1	To what extent do you change configuration of ASATs?	MC	55
Q2.2	Do you use different configurations when working (i) in CI, (ii) Code Review, (iii) locally? If so, why?	O	37
Q2.3	While configuring, do you pay attention to different warnings (i) in CI, (ii) Code Review, (iii) locally?	O	12
Q2.4	Even if you don't configure them, do you pay attention to different warnings (i) in CI, (ii) Code Review, (iii) locally?	O	27
Q2.5	To what extent do you integrate warnings suggested by ASATs during CI?	MC	54
Q2.6	To what extent do you integrate warnings suggested by ASATs during Code Review?	MC	52
Q2.7	To what extent do you integrate warnings suggested by ASATs locally?	MC	50

3.2 Adoption of ASATs

Most of the respondents (48%) declared to use ASATs multiple times per day, while 23% use them on average once per day. As shown in Fig. 2 the most used ASATs are FindBugs (2019) (14%), Checkstyle (2019) (14%) and PMD (2019) (12%). Then, ESLint (2019) and SonarQube (2019) are preferred respectively by 10% and 9% of our

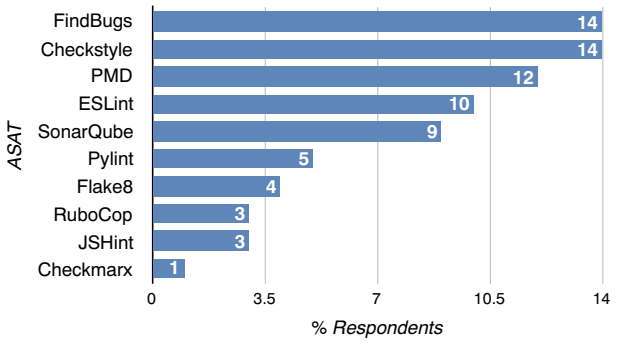


Fig. 2 Top-10 ASATs used by our participants

respondents. Few participants mention other tools, e.g., PYLINT (2019), JSHINT (2019), FLAKE8 (2019), CHECKMARX (2019), and RUBOCOP (2019).

To get a differentiated picture of the ASATs that are frequently used by our participants, we group them by the types defined in Section 2.1.1. We decided that holistic ASATs like SONARQUBE, which can be assigned to more than one type, are counted multiple types. The result is shown in Fig. 3.

Most of our respondents use ASATs to review if variables or methods are correctly named (*Naming*) and to identify error in the exception handling of their applications (*Exceptions*). Other popular choices are to use ASATs to measure code metrics like cyclomatic complexity to ensure *Maintainability* and the adherence to predefined coding standards (*Style*). Less popular, but also reported by our respondent, ASATs are used to check for vulnerabilities (*Security*), to verify *Correctness*, to find potential bottlenecks (*Performance*), or to find *Concurrency* errors. Only 3% of our respondents mention that they use ASATs to detect problems with their *SQL* queries.

The participants who regularly use ASATs (i.e., multiple times per day, or once per day) also indicated the development activities (e.g., bug fixing, refactoring, etc.) during which they usually adopt the tools (Q1.2). We performed a closed card sorting (Spencer 2009) of the described development activities to identify the development contexts in which developers use ASATs. This information enables us to answer RQ_1. Our sorting procedure consisted of four steps:

– We chose two authors as *sorters*, while a third author organized the sorting task. The third author illustrated the sorters (i) the goal of the sorting task, (ii) the conceptual difference between *development activity* (i.e., a task performed by developers working on a project) and *development context* (i.e., a step in the development workflow where some tasks are performed), and (iii) the differences between the proposed development contexts (as described in Section 2).
– The two sorters independently assigned each development activity provided by the respondents (i.e., the cards) to one (or multiple) of the proposed development contexts or (if possible) to *a new context*. The sorters also had the opportunity to say whether a provided activity was not valid (e.g., it was too general to be treated as a real development activity).
– We computed Krippendorff's alpha (Krippendorff 2004) to determine the interrater reliability of the results of the first independent card sorting.

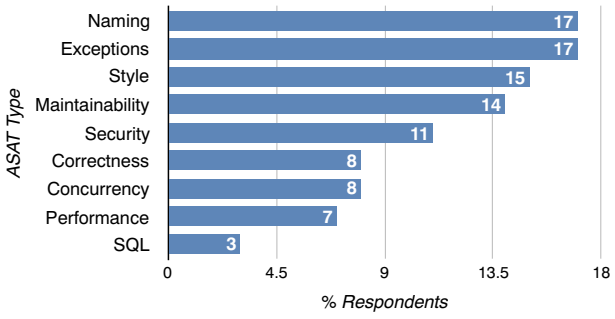


Fig. 3 ASAT types used by our participants

– We involved again the author that set up the sorting task to resolve the conflicts (i.e., the cases where the two sorters partially agree or disagree) and to avoid any bias related to the subjectivity of the sorting.

To not interfere in the card sorting, we decided to not merge activities indicated by our respondents at the beginning. However, in some cases, they clearly refer to the same context. This is the case of "In-Editor typing" and "In-IDE typing": several participants who adopt ASATs during local development indicated that they mainly use ASATs "while implementing the code in the IDE". On the contrary, another participant stated that s/he uses ASATs "while working in the editor". Thus, it is likely that the latter programmer develops using an editor rather than an IDE. Although both types of answer clearly refer to the same activity, we preferred not to merge them to keep the card sorting as clean as possible.

The results of card sorting are shown in Table 5. Our sorters discarded (i.e., marked as not valid) four activities they considered as too generic (e.g., "before a deadline") or not as real activities (e.g., "checkstyle"). Out of the reported 13 activities, the sorters fully agreed on 9, partially agreed on 4, and they never completely disagreed. We computed Krippendorff's alpha coefficient to assess the reliability of the performed sorting. With a score of 0.68, it shows an acceptable agreement (Krippendorff 2004). To summarize, the reported activities could be completely mapped to our initial set of development contexts and it was not necessary to add a new entry in the development contexts we considered in Section 2. Moreover, from the results of Q1.5 we found that 37% of our participants rely on them in CI, 29% in CR and 31% in LP.

Observation 1: ASATs are used by developers in three main contexts: Local Programming, Code Review and Continuous Integration.

To gain further insights into the adoption of ASATs in various contexts, we asked the participants for the reasons of using ASATs individually or in combination (Q1.4). An

Table 5 Results of the closed card sorting applied to the development activities where ASATs are integrated

		Dev. Context			
Activity Name	# Resp.	LP	CR	CI	Agreement
Code Maintenance	4	✓	✓	✓	Full
Code Reviewing	18		✓		Full
CI Build	10			✓	Full
In-Editor typing	1	✓			Full
Pre-commit	4	✓	✓		Partially
Pre-push	4	✓			Full
Build cycle	1			✓	Full
Refactoring	4	✓	✓		Partially
Jenkins stage	1			✓	Full
Debugging	2	✓			Partially
Documentation	1	✓			Partially
Quality Check	3	✓	✓	✓	Full
In-IDE Typing	3	✓			Full

important reason to combine several ASATs seems to be that they *"cover different areas"*, i.e., different rulesets (Buckers et al. 2017). For instance *"Checkstyle helps to detect general coding style issues, while with PMD we can detect error-prone coding practices (including custom rules). FindBugs helps to detect problems which are more visible at bytecode level, like non-optimal operations & resources leaks."*. Another reason is that *"ASATs are language-specific and developers sometimes deal with multiple programming languages in the same project"*.

Interestingly, six participants reported as main motivation for using multiple ASATs the fact that different types of ASATs are needed in different contexts. Specifically:

"[we choose an ASAT] depending on the context. For instance in CR I mainly use Findbugs and PMD.".

In particular, they seem to need ASATs covering different rule sets, as reported by one of the respondents:

"[We install different ASATs] because more tools give more warnings and we can filter these warnings based on style problems (mainly in code reviews) or bugs and other problems possibly breaking compilability (mainly in CI)".

Those initial results about the importance of the development context in the selection of ASAT warnings will be further investigated in Section 4.

3.3 Configuration of ASATs

Beller et al. (2016) have shown that developers tend to adopt ASATs as-is, without evolving or modifying their default configurations. While they have mined this result from software repositories, our RQ_2 was focused on analyzing ASATs configuration from a qualitative point of view.

The results of this analysis are shown in Fig. 4. The general findings by Beller et al. (2016) are confirmed: indeed, more than half of the participants (56%) report that ASATs are configured only during the project kick-off. However, a small but not negligible percentage declared to evolve the tools' configurations on a monthly basis (20%).

To better investigate the motivations behind updating the configuration, we asked whether developers tend to configure ASATs with the aim of adapting them to a specific development context. Most of the respondents (78%) do not use different configurations and they *"forbid configuring static analysis tools as much as possible"* because developers *"want to work with the end-state in mind"* or because it is *"time-consuming to enable/configure them"*. Thus, developers do not use development context for configuring ASATs differently.

Observation 2: Most of the developers do not configure ASATs depending on the development context.

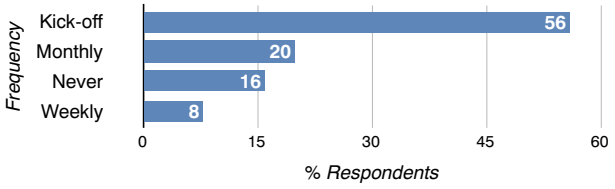


Fig. 4 When ASATs are configured

Despite this general trend, a considerable portion (22%) of our respondents configure ASATs differently depending on the context. Specifically, some of the reasons are:

"When reviewing I want to check the quality of code, when working on my own laptop I want to avoid committing bugs, while style and error checks during CI"

and

"Locally I do not apply any particular configuration, while I like specialized version of the configuration file for continuous integration and code reviews (they require more quality assessment)."

This 22% of our participants claiming to configure ASATs were also surveyed to ask whether they pay attention to different warnings while setting up the tools in different contexts. Some respondents found it hard to answer even though they provided us with some initial insights going in the direction of monitoring different warnings (*"for instance in CI we check translations for issues, check images for being consistent et cetera."*).

On the other hand, we asked participants that do not configure ASATs to think about the types of warnings they usually pay attention to in different contexts (Q2.4). Interestingly, some of the participants said that *"Style warnings are checked during CR, warnings about possible bugs during CI"*, they are *"less worried about pure style issues when developing locally"*, and *"warnings might be not useful in different circumstances [or development contexts]"*. Thus, even though they do not configure ASATs, they tend to use them differently in the various contexts. From these insights we learned that, even though the practice is not wide-spread (as indicated by 78% of our respondents), some developers might need or want to configure ASATs differently depending on the development context. We further analyzed the impact of development contexts on the configuration of ASATs in Section 4.

Finally, from the results of Q2.5-Q2.7 it is important to remark that in all the three development contexts developers rarely ignore the suggestions provided by the ASATs.

4 The Impact of Development Contexts on the Configuration of ASATs

Based on some answers that we received in the context of RQ_1 and RQ_2 the development context can play a role in the configuration of ASATs. As introduced in Section 2.2, the *goal* of this second study is to further investigate this initial finding and analyze how development contexts can influence the selection of warnings (RQ_3). To this end, we studied the developers' opinions on the usage of ASATs and on relevant warnings in different development contexts. The *context* of the second study consists of (i) *subjects*, i.e., the participants to our extended questionnaire, as well as the industrial practitioners interviewed, and (ii) *objects*, i.e., the ASATs used in the analyzed development contexts. The interviewees are numbered S1 to S11. In this section, we describe the overall design of this second study and the results achieved for the two investigated aspects, i.e., factors influencing ASATs usage and relevant warnings in different contexts.

4.1 Study Design

The methodology of this experiment is split into two parts: the design of our extended questionnaire and the design for the semi-structured interviews that we have conducted with professional developers.

Extended Questionnaire As described in Section 2, we extended our initial survey by including additional questions about CONTEXT-BASED USAGE (see Table 6). We focused

Table 6 Added survey questions related to the context-based usage of ASATs (O: Open Question, S: Statement, #: the number of respondents answering the corresponding question)

Section	Summarized Question	Type	#
Context-Based Usage			
Q3.1	Which are the main factors you consider when deciding the set of warnings to look at during Continuous Integration?	O	39
Q3.2	Which are the warning types that are more likely to be fixed during Continuous Integration?	O	39
Q3.3	Which are the main factors you consider when deciding the set of warnings to look at during Code Review?	S	39
Q3.4	Which are the warning types that are more likely to be fixed during Code Review?	S	39
Q3.5	Which are the main factors you consider when deciding the set of warnings to look while working locally?	S	38
Q3.6	Which are the warning types that are more likely to be fixed while working locally?"	S	38

on two main types of questions: (i) what are the factors driving developers' decisions to the selection of the warnings in the three considered contexts (Q3.1, Q3.3, Q3.5) and (ii) what are the warnings they pay more attention to in such contexts (Q3.2, Q3.4, Q3.6).

We have presented an initial list of likely reasons for the usage of ASATs in different contexts to our participants to encourage them to brain-storm about the actual motivations. Dillman et al. (2014) have shown that this methodology stimulates an active discussion and reasoning, thus helping researchers during the investigation of a certain phenomenon. Our proposed list consisted of five factors, i.e., (i) severity of the warnings, (ii) internal policies of the development team, (iii) application domain, (iv) team composition, and (v) tool reputation. These factors have been selected from related literature (Kim and Ernst 2007; Ruthruff et al. 2008) and from the popular question and answer sites StackOverflow (e.g., StackOverflow 2017a, b) and Reddit (e.g., Reddit 2017a, b), which are among the top discussion forums for developers (CryptLife 2017). In the latter case, two of the authors of this paper manually identified likely motivations that push developers into using ASATs in different ways from the developers' discussions.

Semi-Structured Interviews We created an interview guide for our semi-structured interviews to make it easy to keep track of our participants current and past experience with ASATs and to allow them to express their opinions about context based warnings. The guide was split into three sections. In the first section, Background, we asked for years of experience, study degree, programming languages used, role in the company together with its size/domain, and in which development contexts our interviewees adopt ASATs. The second section called Contexts Understanding investigated processes to review and build new software and asked about the different development contexts that exist in the organization. Furthermore, we needed to know how developers use ASATs. In the last section, Usage of ASATs in each context, we asked our interviewees to state which differences they perceive in the usage of ASATs between the different contexts. Furthermore, we intended to extract the *factors* (e.g., size of the change) they take into account while deciding the warnings to look at in each context.

4.2 Main Factors Affecting the Warning Selection

Figure 5 shows the main factors for warning selection as answered by the interviewed developers. The bars show how often a warning type was stated (in percentage) for each development context. The first thing that leaps to the eye is represented by the importance given to the *Severity of the Warnings*. This result confirms that developers mainly rely on the prioritization proposed by the ASATs, and in particular to the proposed levels of severity (e.g., crucial, major, minor) for the selection of the warnings. Developers seem to select the warnings on the basis of their severity, for example postponing the warnings that represent "minor issues" that can be postponed (S9). Our respondents also highlight that it is vital for tools vendors to establish a clear strategy to assign severity because developers *"need to trust the tool in terms of severity"* (S3) and *"it's important to assign the right severity to the rules/warnings"* (S4). In CI the entire build process can fail because of the severity assigned to a warning, *"If there are critical violations, the build fails"* (S2).

While the severity assigned by ASATs plays the most relevant role in the decision process, it is also important to highlight that the surveyed developers pointed out other factors contributing to it. For instance, they highlight that the *policies of the development team* notably influence the way they use ASATs. More specifically, monitoring specific warnings might enforce the introduction of new policies in a team. Indeed, as reported by S7, using ASATs seems to be a *"social factor"*. For example, when a development team decides to adopt a strict policy regarding the naming conventions, it is better that a third party entity reminds a team member when she is not following the established policy. Before starting a project, it is crucial to define a policy in terms of programming standards that should be followed by the entire development team. As pointed out by S10 and S11, ASATs support young team members to follow them. However, as confirmed by S1 it is almost impossible

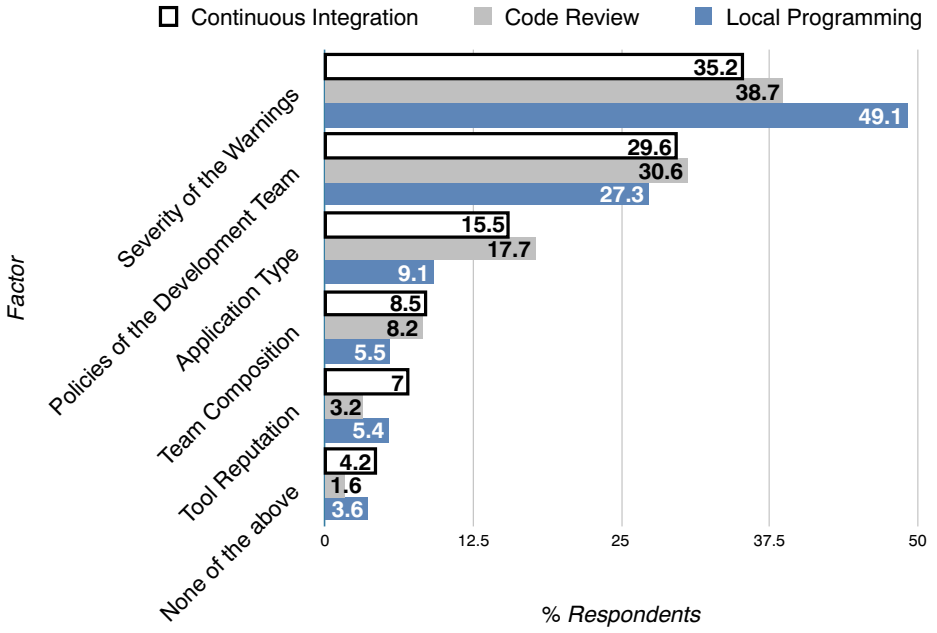


Fig. 5 Main factors while selecting warnings in different contexts

to impose the adoption of specific warnings to developers. Rather, the warnings to monitor have to be somehow *"negotiated with developers"* in the development team, even though in some cases they are erroneously established by the stakeholders, as reported by S2 and S5.

Application Type is the third factor used by our survey participants to select warnings along the different contexts. In particular, an application could be categorized according to its destination, e.g., web service, mobile app, or its lifetime expectation, e.g., long/short term project. According to S1 and S2, the choice of the monitored warnings depends on the application type, which is definitely a key factor to consider. Moreover, S3 also said that *"a short-term application does not need to follow strict rules as the ones related to code structure because they do not need to be maintained for a long time"*.

Still, *Team Composition* represents another factor to take into account. As explained by S3 it *"affects the selection of the warnings because a certain degree of knowledge is needed to understand specific warnings such as SQL injection flaw"*. In other words, some respondents find such warnings hard to integrate in case they do not have teammates having enough expertise for fixing them. However, those warnings can be easily understood if the ASATs provide exhaustive descriptions (Johnson et al. 2013) and possibly propose quick fixes. Thus, *Team Composition* is not so popular among our participants because if the chosen ASAT provides enough support in terms of understandability, every kind of warning can be selected independently from the expertise of the team.

Only a minority of our respondents see the *Tool Reputation* as a crucial factor for warning selection. It is important to remark that, given the nature of our survey study, tools reputation still refers to what developers' perceive as relevant, i.e., we did not quantitatively compute the reputation of tools but relied on the developers' opinions explaining their decisions. However, one of our interviewees (S3) considered it very important since *"developers sometimes do not trust ASATs, because there are no other people that sponsored them"*. It seems that developers need to build up trust and confidence in specific ASATs, but it is not perceived as a key factor for the warning selection.

Finally, one of our respondents highlights the presence of a factor different from the proposed ones. Specifically, he pointed out that *"cost of fixing"* is a key factor for the warning selecting. Indeed, the expected time/effort is important because, when a deadline is approaching, developers might want to postpone issues that do not have a strong impact in the short-term (e.g., style conventions).

Observation 3: Severity is still the most important factor to take into account during the selection of the warnings, even though other factors, e.g., policies of the development team and team composition, play a non-marginal role in the decisional process.

4.3 Different Warnings in Different Contexts

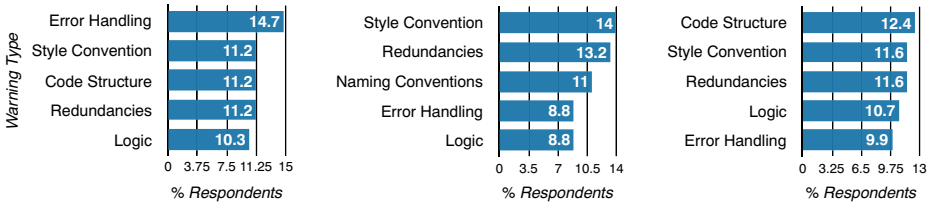
With the aim of comparing the importance developers give to warnings in the different development contexts, our respondents were asked (Q3.2, Q3.4, Q3.6) to indicate which warnings types they usually focus on. To make our results as independent as possible from specific ASATs, we adopted the *General Defect Classification* (GDC) proposed by Beller et al. (2016) as the list of warnings types.

Figure 6 illustrates the warning types that our respondents selected from the GDC in the different contexts. Note that we normalized the data according to the *min-max algorithm* (Al Shalabi et al. 2006) in order to better explain to what extent each warning type is monitored

Fig. 6 Normalized actionability of different warning types

in each context by our participants. Moreover, to point out the warning types that are mostly checked in each development context we factor out the top 5 warnings for CI (Fig. 7a), Code Review (Fig. 7b) and Local Programming (Fig. 7c). In the following, we describe the most relevant categories our participants reported us.

Style Convention is the category concerning typical code style defects such as bad code indentation, missing spaces or tabs. Generally, it is an important category of warnings both in CI (second most selected in Fig. 7a) and locally (second in Fig. 7c), but specifically during code review: it is the warning type selected by the majority of our respondents, as shown in Fig. 7b. This result confirms findings of previous work (Beller et al. 2014; Panichella et al. 2015) that showed that modern code reviews mainly fix design-related issues rather than functional problems. Indeed, S7 reported that the first goal of code review is to verify the adherence to code standards improving sthe code understandability. S9 and



(a) Continuous Integration (b) Code Review (c) Local Programming

Fig. 7 Top-5 warnings to be fixed in different development contexts

S10 confirm during the interviews that style-related issues are crucial points to address during code review. Furthermore, S9 considered it also very valuable while working locally.

Redundancies concern redundant pieces of code or artifacts that can be safely removed. These issues are perceived as very important during code review (the second most important among the most selected warnings) and locally, but also in CI to a lower extent. In particular, S1 confirms the importance of monitoring this category of warnings in CI.

Our respondents also pointed out that they mainly look at *Naming Conventions* during code reviews (third most selected warnings in Fig. 7b), while we have no evidence of this category neither in CI or locally.

Error Handling is the most selected warning in CI, i.e., it occupies the first position among the chosen warnings. It is less important in code review (the fourth most voted in Fig. 7b) and locally (the fifth most voted in Fig. 7c). Indeed only S1 and S3 monitor this category type during code reviews, while most of the interviewees rely on the CI server to spot such issues.

Code Structure reaches the first position in the warnings that are likely to be fixed locally (Fig. 7c). This category concerns rules aiming at checking the structure, in terms of the file system or the coupling, for violations of common conventions. Usually, developers organize the structure of a project locally, so the code structure category is not surprisingly also important for our respondents while working locally. However, our participants tend to not monitor *Code Structure* warnings in code review.

Finally, the *Logic* warnings that are concerned with comparisons, control flow, and algorithms are mostly checked during local programming while they are not considered crucial in CI and code review.

Observation 4: Apart from style conventions that are frequently considered in all the contexts, the perceived importance of warnings is different in the development contexts. When programming in the IDE, developers mainly focus on code structure; when performing code reviews they mainly look at style conventions and redundancies; during CI, they watch handling errors.

5 The Relevance of ASATs in Open-Source Projects

The *goal* of our third study is to investigate the relevance of ASATs in open-source projects (RQ_4 and RQ_5). Differently to previous work of Beller et al. (2016), we do not simply approximate the popularity of ASATs by looking at the presence of ASAT-related files among the projects' resources or surveying projects' contributors, but we review the contribution guidelines of the projects instead and compare them to the configuration that we find in the repository. We compare the definition of ASATs with the projects' contribution guidelines.

Such contribution guidelines form the foundation for shared work on an open-source project. The community defines in a collective effort how ASATs are used to achieve for example certain quality goals or strategies for risk mitigation, by using ASATs. These guidelines should then be considered not only by new contributions to a project (e.g., ASATs can prevent new contributors from making common mistakes resulting in rejected pull

requests Gousios et al. 2015) but also for contributions of existing project members. One part of our investigation has focused on the question of whether the usage of said ASATs are actually enforced in the workflow.

The *context* of our study includes (i) as *subjects* the developers contributing to the inspected open-source projects and (ii) as *objects*, the ASATs used in the 176 open source projects that we manually inspected. In the following sections, we describe the design of the study and the results we obtained.

5.1 Study Design

Our study design consists of the procedure that we adopted to inspect the 176 open source projects (selected as described in Section 2.3). We were interested in investigating to what extent ASATs are defined in open source projects and how their use is enforced while contributing to a project.

To measure the definition of ASATs in open source projects, we examined the projects' repositories. Similarly to previous work (Beller et al. 2016), we searched for ASATs' configuration files (e.g., `google_checks.xml` in case of Google coding conventions for CHECKSTYLE 2019) in the repository or for the explicit declaration of ASATs dependencies in the build configuration file. Considering the most popular ASATs listed in Section 3, we referred to the official ASAT documentation in order to understand how such tools are typically defined and which configuration files are needed. Thus, for each ASAT, we compiled a list of corresponding configuration files (the full list is available in our replication package Vassallo et al. 2019) and we automatically looked them up in the repositories. Developers can define ASATs in arbitrary ways (e.g., they can use a non-default name for the configuration file), so we manually inspected the projects for which the ASATs detection had a negative result. In particular, we searched for files containing the definition of rules and we read the build configuration files in order to reveal the definition of ASATs (e.g., among the build steps or *goals* in Maven 2019). In addition to that, we used the GITHUB `find` function to search for ASATs related terms like "lint", "style", "sonar".

The second part of our inspection procedure regards the ASATs enforcement. To conduct such an investigation, one author inspected the available documentation in the repositories in order to retrieve the contribution guidelines, i.e., the rules that all potential contributors have to live by. Apart from pointing developers to important resources like the issue tracker system and discussion channels (e.g., forum, mailing list, etc.), contribution guidelines include templates for reporting bugs or enhancements, a code of conduct, and requirements for submitting a change. For example in the case of the pull-based software development encouraged by GITHUB (Gousios et al. 2014), a pull-request gets accepted if specific requirements for submitting a change (such as all tests have to pass) are met. If we focus on ASATs, contribution guidelines can enforce the usage of a particular ASAT to perform specific code checks (e.g., code complexity must be below a given threshold) that a change (e.g., submitted as a pull request) must pass in order to have the contribution accepted. Let us consider the contribution guidelines of the `stympy/faker` project[4]. The project maintainers specify that the ASAT called RUBOCOP (2019) has to be used while submitting a pull request. In particular, it is required to invoke a command like `'bundle exec rak'` to *"run the test suite and after that [run] the Ruby static code analyzer"*. Only after passing all the defined RUBOCOP checks, a pull request can be submitted by a contributor. Typically

[4]https://github.com/stympy/faker/blob/master/CONTRIBUTING.md

the contribution guidelines are illustrated in a dedicated file called `CONTRIBUTING.md`. However, during the inspection of a few projects, we found cases where this file did not exist or simply did not describe the guidelines properly. Because of that, we decided to also include other sources in our inspection: the `README.md`, which typically contains instructions on how to install and use the software, and the project (Wikis 2019), which is often used to host further documentation about a project. We carefully studied the contribution guidelines to understand whether ASATs usage is enforced and –if yes– for which types of ASATs. To validate the results of our inspection, an external validator inspected again randomly-selected and statistically-significant sample (with a confidence level of 95% and a confidence interval of $\pm$ 10) of projects for each language. Then, we computed the agreement on both enforced ASATs and code checks in the resulting 122 projects. In case of enforced ASATs, the two inspectors agreed on 104 projects reaching a strong level of inter-rater agreement (Cohen's Kappa (k) (Cohen 1960) of 0.74 that reveals strong agreement[5]). In case of suggested code checks, they agreed on 104 cases with again a strong inter-rater reliability (Cohen's Kappa (k) (Cohen 1960) of 0.71 that means strong agreement[5]). These results make us confident that our inspection results are reliable.

5.2 Definition of ASATs in Open-Source Projects

We evaluated the definition of ASATs by performing an automatic and manual analysis of the projects' repositories. By searching for the presence of configuration files of the most popular ASATs, we were already able to automatically identify the definition of ASATs in 94 projects. After manually inspecting the build configuration files and the projects' repositories, we were able to classify another 23 projects as ASATs-defining projects. As shown in Table 7, open-source projects very frequently define ASATs. 117 systems, corresponding to 66% of the total set of projects, define at least one ASAT in their repository. This percentage is even higher than the one found by Beller et al. (2016) and reveals how the popularity of ASATs has significantly increased over the last 2 years since their study. Grouping projects by language, the percentage of ASAT-defining projects is even higher in the case of Javascript and Python (respectively 81% and 77%). The projects written in Ruby are in line with the average percentage (64%), while Java projects exhibit a lower number (23 corresponding to 47%), but still higher when compared to the results obtained by Beller et al. (2016).

If we further group projects by age, contribution, and popularity (as described in Section 2.3.1) we obtain the results shown in Table 8. The definition of ASATs becomes more important or evident with higher levels of maturity of a project. Indeed, projects in the early stage of their development (i.e., with a low number of commits, stars, and contributors) are less likely to define ASATs (in all three categories the percentage of ASATs-defining projects is below 66%). Vice versa, projects belonging to medium and high age categories exhibit higher percentages than the average in Table 7. This seems to suggest that the need for defining ASATs emerges as soon as the project increases in size (both in terms of commits and contributors experience) and importance.

Among the defined ASATs, we did not find any new ASAT compared to the list obtained surveying developers in Section 3.2. Figure 8a shows a graph of the most defined

[5] $0.2 < k \leq 0.4$ is considered fair, $0.4 < k \leq 0.6$ moderate, $0.6 < k \leq 0.8$ strong, and $k > 0.8$ almost perfect (Cohen 1960)

Table 7 The relevance of ASATs in the analyzed open-source projects

Language	# Projects	# ASATs-defining Projects	# ASATs-enforcing Projects
Java	49	23 (47%)	17 (35%)
Javascript	47	38 (81%)	22 (47%)
Ruby	45	29 (64%)	13 (29%)
Python	35	27 (77%)	14 (40%)
Total	176	117 (66%)	66 (37%)

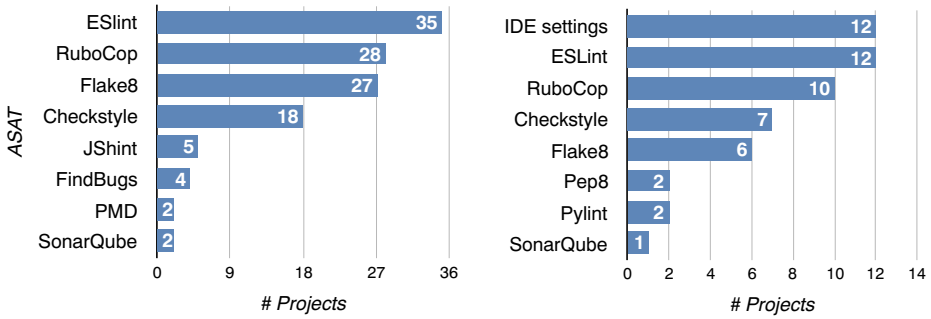
ASATs. Despite the presence of the same ASATs in Fig. 2, the ranking is quite different. ESLINT (2019), RUBOCOP (2019) and FLAKE8 (2019) are frequently defined, while FINDBUGS (2019) and PMD (2019) only in few projects. At the same time, CHECKSTYLE (2019) also seems a popular choice in practice, while SONARQUBE (2019) is not widespread yet. Thus, our results confirm the perceived popularity of certain ASATs, but also that, based on our sample of projects, some ASATs considered less popular than others are instead more frequently defined.

If we group ASATs according to their types (see Section 2.1.1), *Naming* and *Exceptions* are the most defined ASATs (see Fig 9a). At the same ASATs measuring *Maintainability* and spotting *Style* issues are also among the types that are more frequently available in the repositories of our selected projects. It is worth noticing that, in the case of ASATs types, their perceived popularity (see Fig. 3) is in line with their definition. Thus, while the popularity of certain ASAT is not reflected in the reality of open-source projects, the popularity of ASAT types directly matches with their definition in practice.

Observation 5: Developers very often define ASATs in the projects' repositories. The definition of ASATs is higher as long as the maturity, contribution level, and popularity increases.

Table 8 The relevance of ASATs in the analyzed open-source projects grouped by age, contribution, and popularity

Projects Set			ASATs Relevance	
Feature	Level	# Projects	# ASATs-defining	# ASATs-enforcing
Age	Low	44	17 (38.6%)	10 (22.7%)
	Medium	88	62 (70.4%)	29 (32.9%)
	High	44	38 (86.4%)	27 (61.4%)
Contribution	Low	44	13 (29.5%)	10 (22.7%)
	Medium	88	67 (76.1%)	31 (35.2%)
	High	44	37 (84.1%)	25 (56.8%)
Popularity	Low	44	25 (56.8%)	8 (18.1%)
	Medium	88	53 (60.2%)	29 (32.9%)
	High	44	39 (88.6%)	29 (65.9%)

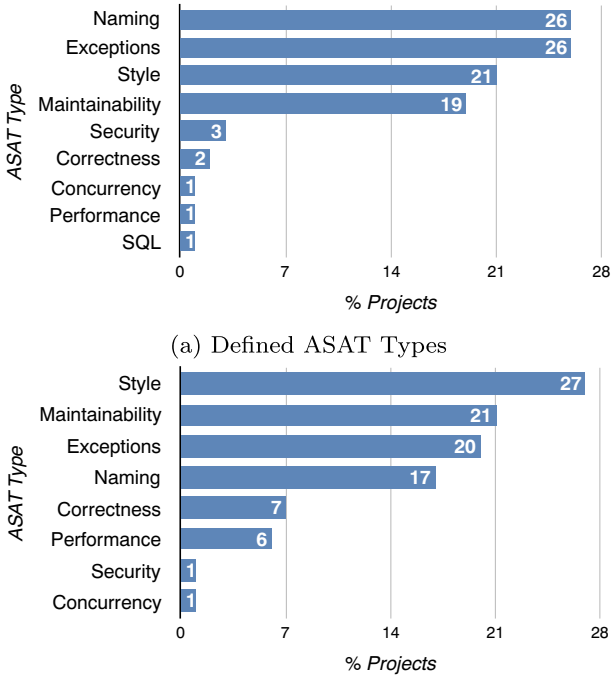


(a) Defined ASATs (b) Enforced ASATs

Fig. 8 The most relevant ASATs in the analyzed open-source projects

5.3 Enforcement of ASATs in Open-Source Projects

Open-source projects very often provide guidelines for potential new contributors. Only 27 out of 176 projects (corresponding to 15%) do not include any contribution guideline. If we



(a) Defined ASAT Types

(b) Enforced ASAT Types

Fig. 9 The most relevant ASAT types in the analyzed open-source projects

consider the remaining 149 projects, only 66 (corresponding to 37% of the size of our sample) enforce the use of ASATs while contributing to a project. Looking at Table 7, Python and Java projects exhibit percentages very close to the average (respectively 40% and 35%), while Javascript projects are more inclined to suggest ASATs (47%). Only 29% of the analyzed Ruby projects enforce ASAT adoption. If we group projects by age, contribution, and popularity (see Section 2.3.1) the pattern that we found in ASATs-defining projects (see Section 5.2) is even more evident. As shown in Table 8 open-source projects enforce more ASATs usage as long as the project evolves. In particular, the percentage of high-popular projects that enforce ASATs is more than double of the corresponding percentage in the case of low-popular projects. If we look at the age and contribution levels, the same pattern holds. If we restrict our attention only to ASAT-defining projects, the overall number of projects that enforce ASATs is slightly smaller (61 compared to 66 in Table 7) and reveals how 52% of those projects that define ASATs enforce their usage. Note that 5 projects are enforcing ASATs, but they are not defining them; these projects encourage new contributors to use the ASAT-capabilities provided by the IDE, or to use an online checker like PEP8 ONLINE CHECK (PEP8 online check 2019). If we further group projects by language, slightly more than half of Java and Javascript projects (∼60%) enforce ASATs usage. For Python, 48% of projects both define and enforce ASATs, while only 41% of ASAT-defining projects written in Ruby enforce them.

To complete our analysis, we wanted to investigate which ASATs are more enforced and for which checks. Figure 8b shows the enforced ASATs in our sample of projects. The most interesting result is about the IDE SETTINGS. We know that the IDE provides typical static analysis features. And looking at the figure, checking code in the IDE is also particularly encouraged by open-source projects suggesting which rules or settings enable. Comparing Figs. 8a and b , ESLINT (2019) and RUBOCOP (2019) are not only frequently defined but also enforced. Obviously, IDE settings are not defined in the projects' repositories. If we distinguish the different ASATs types that have been described in Section 2.1.1, *Style*, *Maintainability*, *Exceptions* and *Naming* are not only the most defined ASATs types but also the ones that are most frequently enforced (see Fig. 9b). However, compared to the results in Fig. 9a *Correctness* and *Performance* types are enforced in more than double of the projects. This might indicate that for some categories open-source projects enforce particular ASATs without defining them in the repository, thus relying on tools that must be configured by contributors on their machine (e.g., use certain plugins in the IDE). With regard to the types of checks, only 58 projects specify which warnings are enabled in the ASATs. *Style Conventions* are by far the most enforced category of warnings across different languages and ASATs. This is in line with our previous results in Section 4, where style conventions are also important across the different development contexts. In 88% of the projects, style conventions are the only reason why the use of ASATs is mandatory. This result confirms that code style is a crucial factor for contributors who want to get a pull request accepted (Gousios et al. 2015). In 10% of the cases coding conventions are followed by licensing (i.e., verify whether the right license header is included in the source code) and in only one case developers are invited to use ASATs to check for bugs and vulnerabilities.

Observation 6: Only a minority of the projects enforces ASAT usage while contributing. Those projects mainly require ASATs for checking code style conventions.

6 The Developers' Perspective on the Relevance of ASATs

The goal of our last study is to complement the quantitative investigation of the relevance of ASATs in open-source development with a qualitative analysis of this phenomenon to understand the developers' perception of the matter (RQ_6). The *context* of this study includes (i) the participants to our forum discussion as *subjects*, and (ii) the specific ASATs used by people commenting to our REDDIT post as *objects*. In the following sections, we describe the design of the study and the results we obtained.

6.1 Study Design

To understand the developers' perspective on the relevance of ASATs in their development process, we posted in the same five popular REDDIT (2019) communities that we have used before (i.e., `java`, `python`, `javascript`, `ruby`, and `learn-programming`). We started a discussion in each community, in which we asked for a reflection on the results of our previous analysis in the context of open-source projects and, in general, on the relevance of ASATs in their development process, we include links to our posts in our replication package (Vassallo et al. 2019).

As discussed in Section 2.4, we gathered a total of 37 comments coming from 29 different subscribers. Our post had the goal of triggering the discussion on the general use of ASATs with a focus on their relevance. Specifically, the post starts with illustrating its goal and presenting the results of our quantitative analysis on the definition and enforcement of ASATs in open source projects. We include two links to the results of our analysis, which refer to the CSV and PDF report of our findings, as available in our replication package. The main results of our investigation are also summarized directly in the post to stimulate readers to reflect on them and provide additional insights. Then, we ask the readers whether and how they use ASATs in their development contexts. One of the authors acted as moderator of the discussion. In particular, this task consisted of (i) answering questions about the study conducted on open source projects (see Section 2.3), e.g., clarify the methodology adopted to perform the inspection, and (ii) replying to comments when some details were ambiguous (e.g., the expression "bunch of metrics" does not clarify which are the actually considered metrics) or when the respondents were too concise (to stimulate them to tell us more). The moderator never judged answers in the discussion and was only responsible for clarifying details or to ask for additional information in case of surprising statements.

After one week of discussion, we perform a card sorting (Spencer 2009) to analyze the received comments. We discarded 8 comments that were lacking any informative content. For instance, we exclude a comment where a subscriber questioned the ability of the subscribers to answer our question, i.e., *"It's a topic that's generally over the heads of the normal audience here"*.

Because of their nature, comments to our post sometimes covered different and controversial aspects of the ASATs usage, e.g., pros and cons of using ASATs. Thus, as first step we decompose the received (and valid) comments in 38 cohesive paragraphs (or cards). Then, we group common cards, and finally organize them hierarchically. The sorting of such cards leads to the 8 main arguments shown in Table 9. In the following we discuss the main findings quoting, when it is needed, extracts from the comments.

Table 9 Main topics resulting from the sorting of REDDIT comments

Topic	# Cards
ASATs ease manual activities	5
I select specific rules to enable	4
I combine different ASATs	4
I do not need ASATs	4
ASATs are buggy	4
ASATs are difficult to configure	7
I use ASATs because my colleagues use them	6
ASATs violations should break the build	4

6.2 ASATs Ease Manual Tasks and Encourage Good Practices within Teams

When our participants refer to their general use, ASATs are mainly perceived as tools that replace humans in performing tedious tasks. Our participants confirm the results of previous work (Panichella et al. 2015) on the potential of using ASATs to increase the automation of tasks that are typically performed manually like code review. ASATs are also important to ease refactoring activities. Our results partially confirm findings of previous work (Vassallo et al. 2018) where ASATs are perceived as tools that both suggest refactoring tasks and provide support for performing them. Several participants pointed out the importance of "setting up [ASATs] to break the build when quality thresholds are not met" (Beller et al. 2017a). This indicates that the involved developers seem to perceive the conceptual violations provided by ASATs at a similar level of severity as other typical reasons for build failures, like compilation errors and testing failures (Vassallo et al. 2017). Furthermore, ASATs have the ability to "encourage good practices" in a development team, thus confirming their importance in spreading the adoption of agreed policies (see Section 4). Some participants remarked that in their team developers use the same tools and this helps in avoiding any possible issues related to the consistency of the changes made to a shared code base.

6.3 Enabling Different Warnings and Combining Different Tools

Developers also see the importance of using ASATs for checking violations different from code style conventions, that might also be mitigated by using code formatter like PRETTIER (2019). Although some of our participants still consider code style violations the most interesting features provided by ASATs (somehow providing a practical motivation behind our findings on the enforcement presented in Section 5.3), their adoption is motivated by other types of checks. They use ASATs to spot memory leaks and avoid possible null pointer exceptions or to perform more advance checks like spotting "possible synchronization bugs and race conditions". Finally, our participants find useful the combination of several ASATs because each of them is specialized in certain types of issues. For example, in Java, they rely on CHECKSTYLE (2019) to enforce code conventions and on FINDBUGS (2019) to spot bugs. This goes pretty much in the direction of one of the emerging (although not widespread yet) ASATs, namely SONARQUBE (2019). It aims at supporting multiple languages and types of checks by aggregating rules from several established ASATs as CHECKSTYLE and FINDBUGS.

6.4 Low Reliability

Some of our participants do not trust the results coming from ASATs. Previous work (Johnson et al. 2013) reveals how developers tend to not use ASATs because of the high rate of false positives. On the one hand, our participants confirm that ASATs generate "a lot of noise" with many warnings that most of the time are irrelevant for the actual contexts where developers are operating. Developers want to be notified regarding their mistakes and not about violations related to code written by others. As a result, especially when they start maintaining a pre-existing system, they disable warnings that affect such "old" code. On the other hand, they report the poor effectiveness of some ASATs in catching issues. For example, some participants complain about FLOW (2019), a static type checker for Javascript code, that sometimes "misses obvious bugs" in the first place. Only after several other changes, FLOW starts finding these latent bugs. Some developers even find the rules quite subjective and difficult to generalize.

6.5 Configuration-Related Difficulties

The configuration of ASATs is perceived as a serious difficulty by our participants. Assuming that team members converge on a set of rules, it is all but trivial to configure ASATs. It is very time-consuming to go through the list of warnings and decide which are close to the team's rules. Thus, our participants tend to rely on the standard configuration confirming the results of previous work (Beller et al. 2016) and the ones presented in Section 3. Despite developers finding it useful to combine different ASATs, some tools are not compatible. One participant has mentioned the example of PYLINT (2019) and FLAKE8 (2019) in Python code. She would like to have both installed and run them at every new change. However, in the case of *unused variables*, PYLINT suggests developers write "unused_ in front of [them]", while FLAKE8 recognizes that action as a code style violation. At the same time, it is not so easy to propose enhancements or receive support while using ASATs. The ASATs communities are not very open and in most of the cases not responsive to change requests.

Observation 7: Developers acknowledge the potential of ASATs, but their perceived "bugginess" and challenging configuration limit the ASATs adoption in practice.

7 Discussion

In this section, we discuss the main findings of our study and their implications for researchers and practitioners.

For RQ_1 we found that developers adopt ASATs while working in the IDE, reviewing code made by other developers or simply building new software releases. All those tasks flow into three main development contexts, i.e., local programming (LP), code review (CR), and continuous integration (CI). The usage of ASATs is almost equally distributed among the contexts: 37% of our survey participants rely on ASATs while integrating code changes in an existing project, 29% while reviewing code and 31% while working locally.

Conclusion 1: ASATs are adopted in three main development contexts, i.e., local environment, code review and continuous integration.

In RQ_2, we discovered that 56% of the respondents to our survey configure ASATs at least once before starting a new project. This result generally confirms previous findings reported by Beller et al. (2016), who showed that developers did not change ASAT configurations often. Despite its usage in these three different contexts, the majority of developers (78% of our participants) declared to not make a distinction while using ASATs in CI, CR, or LP. The main motivation for which ASATs users do not enable different warnings when switching from one context to another is that they perceive not working *"with the end-state in mind"* as harmful.

Conclusion 2: Developers do not enable different warnings in different development contexts.

When analyzing the factors taken into account by developers to select the enabled warnings, we found that severity is highly relevant. However, it represents *only a part of the whole story* and other factors also play a role. For instance, internal policies of the development team (e.g., the enforcement of specific programming standards or style conventions) or the life expectation of an application.

Conclusion 3: Severity of the warnings is the main factor when selecting warnings, however, there are other factors to take into account.

In RQ_3, we observed that developers usually *pay attention* to different categories of defects while working locally, in code review or rather in CI. Specifically, they mainly look at *Error Handling* in CI, at *Style Convention* in Code Review, and at *Code Structure* locally. These warnings are not mutually exclusive though and some categories appear in different contexts with different weights.

Conclusion 4: The actual ASAT configurations do not reflect the developers' perception of warnings to monitor in each development context.

In RQ_4, we strengthened previous findings (Beller et al. 2016) on the popularity of ASATs in open-source projects. We found that the definition of ASATs in projects' repositories has increased during the last years (66% of our sample of projects integrate ASATs),

However, in RQ_5 we discovered that only half of the projects (corresponding to 37% of our sample) enforce their adoption. Those results mean that developers do not consider the usage of ASATs as mandatory for getting a submitted pull request accepted and more in general for contributing to a project. Furthermore, *Style Conventions* are the only type of warnings for which ASATs are suggested to be used.

Conclusion 5: Developers underuse ASATs. Typically, ASATs are defined in projects' repositories, but only a few projects enforce their usage and just for checking coding style conventions.

In RQ_6, we reinforced our investigation for RQ_4 and RQ_5 by inviting developers to reflect on the obtained results on the ASATs' relevance in open-source projects. We found an agreement on the potential of using ASAT warnings also for non-code style related violations to avoid (tedious) manual tasks, and on the team culture as a factor for suggesting their use. However, the low enforcement can be motivated by looking at the lack of reliability of those tools and the difficulty to configure them according to the developers' needs. ASATs sometimes do not reveal the presence of obvious defects, or simply have an unexpected behavior when integrated with other tools.

Conclusion 6: Developers believe that ASATs have the potential to ease manual tasks and spread good practices. However, their "bugginess" and hard configuration prevent ASATs from being enforced.

Our findings have important implications for both researchers and ASATs vendors:

Biased Perception We have seen a contrast between what developers think about ASAT configurations and what they pay attention to in practice. This suggests the need for future research into novel techniques that can estimate the actual factors that influence the selection of warnings, e.g., metrics that quantify developers' team composition and experience, while ASATs vendors need to provide or integrate additional information besides the severity of warnings to developers.

Holistic Analysis of the Developers' Behavior Our study revealed that there is not a mutually exclusive set of warnings developers focused on in different contexts, even though such warnings have a different relative "weight". Moreover, we found that it is almost impossible to enforce the adoption of specific warnings to developers. These results suggest the need for future research devoted to the implementation of novel tools that are able to estimate good weights for the context-specific warning-selection of ASATs. To this end, telemetry data about developer activities (e.g., Proksch et al. 2017, Dias et al. 2013) might provide useful input for personalized ASAT suggestions and, thus, improve the usability of these tools in practice.

Towards Context-Awareness A clear implication of our results is the need for a new generation of ASATs that are able to improve the user experience of developers using them, by selecting the warnings to fix in a more context-dependent manner. This includes (i) the adoption of novel methodologies able to automatically understand the context in which a developer is working in at a certain moment; (ii) the definition of smart filters/prioritization mechanisms able to learn from context-based historical information how to properly support the adoption of ASATs in each context.

Understanding Factors Making ASATs Underused ASATs provide early identification of defects, vulnerabilities, and typical maintainability warnings as code complexity. The majority of developers consider ASATs as a non-essential tool while contributing to a project. An initial investigation of the possible reasons reveals that ASATs are not only affected by a high rate of false positives (Johnson et al. 2013) but also that they are unable to uncover sometimes obvious bugs and are difficult to be configured. Furthermore, when adopted, the use of ASATs is enforced only for spotting code style violations. Thus, apart from an investigation into the factors leading to the little enforcement in open-source

projects, there is the need to analyze the reasons why some categories of warnings are not enabled. Specifically, the research community needs to figure out whether this is due to poor knowledge of ASATs' capabilities or to a higher rate of false positives generated by specific categories where ASAT vendors need to focus their attention.

Feedback-driven ASAT Rules Several participants mentioned the difficulty to communicate new feature requests or just reporting bugs to ASAT vendors. While this finding needs to be further studied and verified by the research community, this might potentially be one of the reasons for typical issues connected to the use of ASATs like the high false positives rate. ASATs vendors could provide users with the possibility to report live issues, e.g., false positives, while using their tool. Collecting such reports can help ASAT vendors in understanding the circumstances leading a particular warning to be perceived as not relevant by the users. Thus, novel software analytics mechanisms helping ASAT vendors in understanding how their tools are actually used in practice can potentially be the means in which the gap between envisioned and actual usage is brought closer together.

8 Threats to Validity

Threats to *construct validity* concern the way in which we set up our study. Most of the participants performed the two surveys in a remote setting: thus, we could not avoid the lack of conscientious responses or oversee their actual behavior in the various development contexts. Furthermore, the metadata sent to us from study participants could be affected by imprecisions: in some cases, not all questions have been answered or some of them were answered superficially. To mitigate these threats we first shared the surveys using an online survey platform and forced participants to fill in the main questions. Secondly, we complemented the questionnaires by involving 11 industrial experts that use ASATs on a daily basis. Moreover, to complement the results achieved when surveying and interviewing developers, we have analyzed projects on GitHub and manually investigated their contribution guidelines and/or examined the actual incorporation of ASATs as well as the types of warnings for which ASAT controls are usually enforced. All these analyses complement each other and are useful to ensure the reliability of the obtained results, providing a unified view on the usage, relevance and perceived usefulness of ASATs for developers.

By advertising the survey on social media channels such as Facebook and Twitter using our personal accounts we could have introduced some form of selection bias. However, it is important to note that we mitigated this threat in two different manners. On the one hand, we extended the invitations to 52 randomly selected developers coming from OSS projects adopting ASATs and available on TravisTorrent (Beller et al. 2017a), thus using an opportunistic sample approach able to complement the initial selection process. On the other hand, we advertised the survey on Reddit (2019), which is an independent forum where it was possible to focus on experts opinions about the topics of our research. Of course, we cannot still exclude self-selection or voluntary response bias, i.e., who volunteered to respond may be more involved with ASATs than the average developer.

A further threat relates to the relationship between theory and experimentation. These are mainly due to imprecision in our measurements. As for the survey, we used a 5-level Likert scale (Oppenheim 1992) to collect the perceived relevance of some ASAT practices. To limit random answers, we have provided the participants with the opportunity to explain the answers with free comment fields.

Threats to *internal validity* are related to confounding factors that might have affected our results. In the context of RQ_1, the card sorting (Spencer 2009) matching ASAT usage to the correct development contexts was firstly performed by two authors independently, and then a discussion to solve conflicts took place. A third evaluator participated in the discussion to mitigate threats due to the subjectivity of the classification.

Threats to *external validity* concern the generalizability of our findings. In our surveys, we involved both industrial and open-source developers: they also had a very diverse background and come from projects pretty different in terms of domain and size. As for the developers involved in the semi-structured interviews, they had solid development experience. Clearly, it is possible that some of our results partially generalize to other organizations and open source companies. To limit this threat, as mentioned before, we complement the results achieved when surveying and interviewing developers by mining projects on GitHub, thus sampling the top-rated projects (more details in Section 5), related to the main program languages—Java, Javascript, Ruby, and Python—that emerged in the first study (see Section 3). However, it is important to note that the results achieved on the incorporation/enforcement of ASATs might be due to the context selection process we have performed. Hence, further studies are needed to verify our findings: this is especially true when it turns to the incorporation/enforcement of ASATs in the context of industrial environments. Finally, we observe the developers' perception of the adoption of ASATs by creating Reddit discussions (see Section 5). Also, in this case, our findings concern a limited set of participants, i.e., the ones that answered our questions on specific subreddits, and further studies are needed to verify the generalizability of these findings.

9 Related Work

The use of static analysis tools for software defect detection is a common practice for developers during software development and has been studied by several researchers (Flanagan et al. 2002; Wagner et al. 2005; Nagappan and Ball 2005; Zheng et al. 2006; Nanda et al. 2010; Butler et al. 2010). This section discusses the related literature on empirical studies investigating the warnings and the problems detected by static analysis tools in the software evolution (Flanagan et al. 2002; Wagner et al. 2005; Zheng et al. 2006; Heckman and Williams 2011; Beller et al. 2016), code review (Panichella et al. 2015) and continuous integration (Zampetti et al. 2017) development contexts.

In past and recent years, ASATs have captured the attention of researchers under different perspectives. Flanagan et al. (2002) investigated the usefulness of two ASATs, i.e., ESC-Java and CodeSonar, discovering that they have reliable performance. Wagner et al. (2005) evaluated the usefulness of FindBugs, PMD and QJ Pro by analyzing four small Java projects. They found that the tools results varied across different projects and their effectiveness strictly depends on the developers programming style. At the same time, Ayewah et al. (2007) showed that the defects reported by FindBugs are issues that developers are actually interested in to fix. Zheng et al. (2006) evaluated the types of errors that are detected by bug finder tools and their effectiveness in an industrial setting. Results of their study show that the detected defects can be effective for identifying problematic modules. Rahman et al. (2014) statistically compared defect prediction tools with bug finder tools and demonstrated that the former achieve better results than PMD, but worse than FindBugs. Instead, Nagappan and Ball (2005) found that the warning density of static analysis tools is correlated with pre-release defect density. Moreover, Butler et al. (2010) found that, in general, poor quality identifier names are associated with a higher density of warnings reported by FindBugs.

D'silva et al. (2008) and Heckman and Williams (2011) did a survey on the algorithms that perform automatic static analysis of software to detect programming errors or prove their absence in industrial contexts.

Kim and Ernst (2007) studied how warnings detected by JLint, FindBugs, and PMD tools are removed during the project evolution history. Their results show that warning prioritization done by such tools tends to be ineffective. Indeed, only 10% of them are removed during bug fixing, whereas the others are removed in other circumstances or are false positives. They suggested prioritizing warnings using historical information, improving warning precision in a range between 17% and 67%. A related analysis, focusing on vulnerability, was also performed by Di Penta et al. (2009) who studied what kinds of vulnerabilities developers tend to remove from software projects. In addition, Thung et al. (2012) and Nanda et al. (2010) evaluated the precision and recall of static analysis tools by manually examining the source code of open source and industrial projects. Their results highlight that static analysis tools are able to detect many defects even though a substantial proportion of them is still not captured. These findings were later confirmed by Nanda et al. (2010).

Beller et al. (2016) analyzed nine ASATs, finding that their default configurations are almost never changed and that developers tend to not add new custom analyses. Our work acts as triangulation of these findings: indeed, we could qualitatively confirm that developers tend to modify the default configurations only at the beginning of the project.

The work by Zampetti et al. (2017) and Panichella et al. (2015) were conducted in the context of continuous integration and code review, respectively. The former showed that a small percentage of the broken builds are caused by problems caught by ASATs and that missing adherence to *coding standards* is the main cause behind those failures. The latter showed that during code review the most frequently fixed warnings are related to *imports*, *regular expression*, and *type resolution*. We share with the study by Panichella et al. the finding that ASATs tools can be useful when properly configured, as developers pay attention to specific warnings during code reviews. However, we substantially highlight how the selection of static analysis tools and warnings vary from different development contexts and this depends on the project culture and developers' ASATs perceived usefulness. Nurolahzade et al. (2009) confirmed the findings by Panichella et al. and showed that reviewers not only try to improve the code quality, but they also try to identify and eliminate immature patches. Our study can be considered complementary to these papers: while Panichella et al. (2015) and Zampetti et al. (2017) focused on single contexts, we propose a more holistic analysis of the developers' behavior over different development stages in order to understand which are the warning types that are most relevant in the different contexts.

Using static analysis tools for automating code inspections can be beneficial for software engineers and the study by Johnson et al. (2013) investigated why developers are not widely using static analysis tools and how current tools could potentially be improved. This study involved interviews with 20 developers and, consistent with our work, highlights that although all of our participants felt that use is beneficial, false positives and the way in which the warnings are presented, among other things, are barriers to use. Compared to the work by Johnson et al. (2013), our paper involves more developers and investigates more development contexts. Furthermore, our paper involves a mix of quantitative and qualitative analysis, thus providing insights into how to properly improve prioritization strategies characterizing current ASAT usage contexts.

Recent work investigated the limits of ASAT tools in industrial (e.g., Google Sadowski et al. 2018) and open source context (Vassallo et al. 2018), and proposed solutions to reduce the number of alarms they generate (Muske et al. 2018; Bodden 2018) or summarize the ASAT-related information contained in build logs (Vassallo et al. 2018). Moreover, Mah-

mood and Mahmoud (2018) compare static analysis tools for Java and C/C++ source code and explore their pros and cons. However, none of these works have investigated the usage of these tools in different contexts or examined their enforcement in open source projects.

10 Conclusion

In this paper, we have investigated the usage of ASATs in practice from two different perspectives. On the one hand, we studied whether developers use ASATs distinctly in different development contexts, i.e., IDE, code review, and continuous integration. On the other hand, we have conducted a study aimed at understanding whether ASATs are defined and enforced in open source projects. As an additional contribution, we also provide qualitative insights on the relevance of ASATs by creating REDDIT (2019) discussions and gathering comments directly from developers that use those tools on a daily basis.

Our study highlighted a number of major findings that may lead to further research on warning prioritization as well as a better organization of warning by ASAT vendors. Specifically, we first observed that (i) developers mainly use ASATs in three different development contexts, i.e., local environment, code review, and continuous integration, (ii) developers configure ASATs at least once during a project, and (iii) although developers do not change the ASAT configuration when working in different contexts, they assign different priorities to different warnings along the contexts. Moreover, we showed a clear limitation of the current state of the practice: while developers of open-source systems generally have ASATs defined, only a few projects enforce their usage and, when they do, this is just for checking coding style conventions. In other words, the potential of ASATs is not fully embraced. This problem represents the core of our future research agenda, which is focused on the definition of novel automated strategies able to help developers in paying attention toward the right warnings depending on the context they are in. Moreover, we aim at further investigating possible strategies to increase the developers' awareness and actual adoption of ASATs in practice. Finally, based on the results of our RQ_3, we also hypothesize the need for different tool types in different contexts (covering different rule sets): thus, as future work, we plan to analyze whether particular categories of ASATs are used in each context.

Acknowledgements The authors would like to thank all the open-source and industrial developers who responded to our survey, as well as the 11 industrial experts that participated to the semi-structured interviews. We also thank Diego Martin, which acted as external validator of the enforced ASATs and types of code checks. This research was partially supported by the Swiss National Science Foundation through the SNF Projects Nos. 200021-166275 and PP00P2_170529, and the Dutch Science Foundation NWO through the TestRoots project (project number 639.022.314).

References

Hovemeyer D, Pugh W (2004) Finding Bugs is Easy. In: OOPSLA 2004, ACM, pp 132–136. http://doi.acm.org/10.1145/1028664.1028717

Al Shalabi L, Shaaban Z, Kasasbeh B (2006) Data mining: a preprocessing engine. J Comput Sci 2(9):735–739

Ayewah N, Pugh W, Hovemeyer D, Morgenthaler JD, Penix J (2008) Using static analysis to find bugs. IEEE Softw 25(5):22–29. https://doi.org/10.1109/MS.2008.130

Ayewah N, Pugh W, Morgenthaler JD, Penix J, Zhou Y (2007) Evaluating static analysis defect warnings on production software. In: Das M, Grossman D (eds) Proceedings of the 7th ACM SIGPLAN-SIGSOFT

workshop on program analysis for software tools and engineering, PASTE'07, San Diego, California, USA, June 13-14, 2007. ACM, pp 1–8. https://doi.org/10.1145/1251535.1251536
Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 2013 international conference on software engineering. IEEE Press, pp 712–721
Balachandran V (2013) Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In: Proceedings of the international conference on software engineering (ICSE). IEEE, pp 931–940. https://doi.org/10.1109/ICSE.2013.6606642
Beller M, Bacchelli A, Zaidman A, Juergens E (2014) Modern code reviews in open-source projects: which problems do they fix? In: Proceedings of the 11th working conference on mining software repositories. ACM, pp 202–211
Beller M, Bholanath R, McIntosh S, Zaidman A (2016) Analyzing the state of static analysis: a large-scale evaluation in open source software. In: IEEE 23rd international conference on software analysis, evolution, and reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016, vol 1. IEEE Computer Society, pp 470–481. https://doi.org/10.1109/SANER.2016.105
Beller M, Gousios G, Panichella A, Proksch S, Amann S, Zaidman A (2017) Developer testing in the IDE: patterns, beliefs, and behavior. IEEE Trans Softw Eng (TSE)
Beller M, Gousios G, Panichella A, Zaidman A (2015a) When, how and why developers (do not) test in their IDEs. In: Proceedings of the joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE). ACM, pp 179–190
Beller M, Gousios G, Zaidman A (2015b) How (much) do developers test? In: 37th IEEE/ACM international conference on software engineering (ICSE 2015). IEEE Computer Society, pp 559–562
Beller M, Gousios G, Zaidman A (2017) Oops, my tests broke the build: an explorative analysis of travis ci with github. In: International conference on mining software repositories. IEEE Press, pp 356–367
Beller M, Gousios G, Zaidman A (2017) TravisTorrent: synthesizing Travis CI and GitHub for full-stack research on continuous integration. In: Proceedings of the 14th working conference on mining software repositories. IEEE, pp 447–450
Bitbucket (2019) https://bitbucket.org/. Accessed: 2019-03-10
Bodden E (2018) Self-adaptive static analysis. In: Proceedings of the 40th international conference on software engineering: new ideas and emerging results, ICSE-NIER '18. ACM, New York, pp 45–48. https://doi.org/10.1145/3183399.3183401
Buckers T, Cao C, Doesburg M, Gong B, Wang S, Beller M, Zaidman A (2017) UAV: warnings from multiple automated static analysis tools at a glance. In: IEEE 24th international conference on software analysis, evolution and reengineering (SANER). IEEE Computer Society, pp 472–476
Bundler (2019) https://bundler.io/. Accessed: 2019-03-10
Butler S, Wermelinger M, Yu Y, Sharp H (2010) Exploring the influence of identifier names on code quality: an empirical study. In: Proceedings of the European conference on software maintenance and reengineering (CSMR), pp 156–165
Catolino G, Palomba F, De Lucia A, Ferrucci F, Zaidman A (2018) Enhancing change prediction models using developer-related factors. J Syst Softw 143:14–28
Checkmarx (2019) https://www.checkmarx.com/. Accessed: 2019-03-10
CheckStyle (2019) http://checkstyle.sourceforge.net. Accessed: 2019-03-10
Chen L (2015) Continuous delivery: huge benefits, but challenges too. IEEE Softw 32(2):50–54
Cohen J (1960) A coefficient of agreement for nominal scales. Educ Psychol Meas 20(1):37–46
CodePro (2019) https://www.roseindia.net/eclipse/plugins/tool/CodePro-AnalytiX.shtml. Accessed: 2019-03-10
CryptLife (2017) Top ten forums for programmers, https://www.cryptlife.com/designing/programming/10-best-active-forums-for-programmers
Di Penta M, Cerulo L, Aversano L (2009) The life and death of statically detected vulnerabilities: an empirical study. Inf Softw Technol 51(10):1469–1484
Dias M, Cassou D, Ducasse S (2013) Representing code history with development environment events. In: International workshop on smalltalk technologies
Dillman DA, Smyth JD, Christian LM (2014) Internet, phone, mail, and mixed-mode surveys: the tailored design method. Wiley, New York
D'silva V, Kroening D, Weissenbacher G (2008) A survey of automated techniques for formal software verification. IEEE Trans Comput Aided Des Integr Circuits Syst 27(7):1165–1178
Emanuelsson P, Nilsson U (2008) A comparative study of industrial static analysis tools. Electron Notes Theor Comput Sci 217:5–21
ESLint (2019) https://eslint.org/. Accessed: 2019-03-10

Findbugs (2019) http://findbugs.sourceforge.net/index.html. Accessed: 2019-03-10
flake8 (2019) http://flake8.pycqa.org/en/latest/. Accessed: 2019-03-10
Flanagan C, Leino KRM, Lillibridge M, Nelson G, Saxe JB, Stata R (2002) Extended static checking for java. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI), pp 234–245
Flow (2019) https://flow.org/ Accessed: 2019-03-10
Gibbs L, Kealy M, Willis K, Green J, Welch N, Daly J (2007) What have sampling and data collection got to do with good qualitative research? Aust N Z J Public Health 31(6):540–544
Gerrit (2019) https://code.google.com/p/gerrit/. Accessed: 2019-03-10
Github (2019) https://github.com/. Accessed: 2019-03-10
Gitlab (2019) https://about.gitlab.com/. Accessed: 2019-03-10
Gousios G, Pinzger M, van Deursen A (2014) An exploratory study of the pull-based software development model. In: Proceedings of the 36th international conference on software engineering, ICSE 2014. ACM, New York, pp 345–355. https://doi.org/10.1145/2568225.2568260
Gousios G, Zaidman A, Storey MA, van Deursen A (2015) Work practices and challenges in pull-based development: the integrator's perspective. In: 37th IEEE/ACM international conference on software engineering (ICSE 2015). IEEE computer society, pp 358–368
Gradle (2019) https://gradle.org/. Accessed: 2019-03-10
Heckman SS, Williams LA (2011) A systematic literature review of actionable alert identification techniques for automated static code analysis. Inf Softw Technol 53(4):363–387. https://doi.org/10.1016/j.infsof.2010.12.007
Hilton M, Tunnell T, Huang K, Marinov D, Dig D (2016) 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). ACM, pp 426–437
Coverity (2009) Effective management of static analysis vulnerabilities and defects. https://pdfs.semanticscholar.org/1970/a4d1746734577a6eb4fdd783668f6b4202ef.pdf. Accessed 20 Aug 2019
Johnson B, Song Y, Murphy-Hill ER, Bowdidge RW (2013) Why don't software developers use static analysis tools to find bugs? In: Notkin D, Cheng BHC, Pohl K (eds) 35th international conference on software engineering, ICSE'13, San Francisco, CA, USA, May 18-26, 2013. IEEE Computer Society, pp 672–681. https://doi.org/10.1109/ICSE.2013.6606613
Johnson RB, Onwuegbuzie AJ (2004) Mixed methods research: a research paradigm whose time has come. Educ Res 33(7):14–26
Johnson SC (1977) Lint, a C program checker. Bell Telephone Laboratories Murray Hill
Jørgensen M (2004) A review of studies on expert estimation of software development effort. J Syst Softw 70(1-2):37–60
JSHint (2019) https://jshint.com/. Accessed: 2019-03-10
Khoshgoftaar TM, Allen EB (1998) Classification of fault-prone software modules: prior probabilities, costs, and model evaluation. Empir Softw Eng 3(3):275–298
Kim S, Ernst MD (2007) Which warnings should I fix first? In: Proceedings of the the 6th joint meeting of the european software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, ESEC-FSE '07. ACM, pp 45–54. https://doi.org/10.1145/1287624.1287633
Krippendorff K (2004) Content analysis: an introduction to its methodology, 2nd edn. Sage, London
Lehman MM (1980) On understanding laws, evolution, and conservation in the large-program life cycle. J Syst Softw 1:213–221. https://doi.org/10.1016/0164-1212(79)90022-0
Mahmood R, Mahmoud QH (2018) Evaluation of static analysis tools for finding vulnerabilities in java and C/C++ source code. arXiv:1805.09040
Maven (2019) http://maven.apache.org/plugins/index.html Accessed: 2019-03-10
McIntosh S, Kamei Y, Adams B, Hassan AE (2014) The impact of code review coverage and code review participation on software quality: a case study of the qt, vtk, and ITK projects. In: Proceedings of the working conference on mining software repositories (MSR), pp 192–201
Muske T, Talluri R, Serebrenik A (2018) Repositioning of static analysis alarms. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, ISSTA 2018. ACM, New York, pp 187–197. https://doi.org/10.1145/3213846.3213850
Nagappan N, Ball T (2005) Static analysis tools as early indicators of pre-release defect density. In: Proceedings of the international conference on software engineering (ICSE), pp 580–586
Nanda MG, Gupta M, Sinha S, Chandra S, Schmidt D, Balachandran P (2010) Making defect-finding tools work for you. In: Proceedings of the international conference on software engineering (ASE), vol 2, pp 99–108
Novak J, Krajnc A, Žontar R (2010) Taxonomy of static code analysis tools. In: The 33rd international convention MIPRO, pp 418–422

Nurolahzade M, Nasehi SM, Khandkar SH, Rawal S (2009) The role of patch review in software evolution: an analysis of the mozilla firefox. In: Proceedings of the joint international and annual ERCIM workshops on principles of software evolution (IWPSE) and software evolution (Evol) workshops, pp 9–18

Oppenheim B (1992) Questionnaire design, interviewing and attitude measurement. Pinter Publishers, London

Palomba F, Zanoni M, Fontana FA, De Lucia A, Oliveto R (2017) Toward a smell-aware bug prediction model. IEEE Trans Softw Eng 45(2):194–218

Panichella S, Arnaoudova V, Di Penta M, Antoniol G (2015) Would static analysis tools help developers with code reviews? In: 22nd IEEE international conference on software analysis, evolution, and reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015, pp 161–170. https://doi.org/10.1109/SANER.2015.7081826

Parnas DL, Lawford M (2003) The role of inspection in software quality assurance. IEEE Trans Softw Eng 29(8):674–676

PEP8 online check (2019) http://pep8online.com/ Accessed: 2019-03-10

PMD (2019) http://pmd.sourceforge.net. Accessed: 2019-03-10

Prettier (2019) https://prettier.io/ Accessed: 2019-03-10

Proksch S, Nadi S, Amann S, Mezini M (2017) Enriching in-ide process information with fine-grained source code history. In: International conference on software analysis, evolution, and reengineering

Pylint (2019) https://www.pylint.org/. Accessed: 2019-03-10

Rahman F, Khatri S, Barr ET, Devanbu PT (2014) Comparing static bug finders and statistical prediction. In: Proceedings of the international conference on software engineering (ICSE), pp 424–434

Reddit (2017a) Php static analysis tools, https://www.reddit.com/r/PHP/comments/5d4ptt/static_code_analysis_tools_veracode/

Reddit (2017b) Static analysis tools, https://www.reddit.com/r/programming/comments/3087rz/static_code_analysis/

Reddit (2019) https://www.reddit.com/. Accessed: 2019-03-10

Rigby PC (2011) Understanding open source software peer review: review processes, parameters and statistical models, and underlying behaviours and mechanisms. Ph.D. thesis, University of Victoria, BC Canada

Rigby PC, German DM (2006) A preliminary examination of code review processes in open source projects. Tech. Rep. DCS-305-IR, University of Victoria

Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empirical Softw Engg 14(2):131–164. https://doi.org/10.1007/s10664-008-9102-8

Rubocop (2019) https://github.com/rubocop-hq/rubocop Accessed: 2019-03-10

Ruthruff JR, Penix J, Morgenthaler JD, Elbaum S, Rothermel G (2008) Predicting accurate and actionable static analysis warnings: an experimental approach. In: Proceedings of the 30th international conference on software engineering. ACM, pp 341–350

Sadowski C, Aftandilian E, Eagle A, Miller-Cushon L, Jaspan C (2018) Lessons from building static analysis tools at Google. Commun ACM 61(4):58–66. https://doi.org/10.1145/3188720

Sadowski C, van Gogh J, Jaspan C, Söderberg E, Winter C (2015) Tricorder: building a program analysis ecosystem. In: Bertolino A, Canfora G, Elbaum SG (eds) 37th IEEE/ACM international conference on software engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, vol 1. IEEE Computer Society, pp 598–608. https://doi.org/10.1109/ICSE.2015.76

SBT (2019) https://www.scala-sbt.org/. Accessed: 2019-03-10

SonarQube (2019) http://www.sonarqube.org Accessed: 2019-03-10

Spencer D (2009) Card sorting: designing usable categories. Rosenfeld Media

StackOverflow (2017a) Static analysis tool customatization, https://stackoverflow.com/questions/2825261/static-analysis-tool-customization-for-any-language

StackOverflow (2017b) Static analysis tools, https://stackoverflow.com/questions/22617713/whats-the-current-state-of-static-analysis-tools-for-scala

Thung F, Lucia L, Lo D, Jiang L, Rahman F, Devanbu PT (2012) To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In: Proceedings of the international conference on automated software engineering (ASE), pp 50–59

Vassallo C, Palomba F, Bacchelli A, Gall HC (2018) Continuous code quality: are we (really) doing that? In: ASE. ACM, pp 790–795

Vassallo C, Palomba F, Gall HC (2018) Continuous refactoring in ci: a preliminary study on the perceived advantages and barriers. In: 34th IEEE international conference on software maintenance and evolution (ICSME)

Vassallo C, Panichella S, Palomba F, Proksch S, Gall HC, Zaidman A (2019) Replication package for "How developers engage with static analysis tools in different contexts". https://doi.org/10.5281/zenodo.3253223

Vassallo C, Panichella S, Palomba F, Proksch S, Zaidman A, Gall HC (2018) Context is king: the developer perspective on the usage of static analysis tools. In: SANER. IEEE Computer Society, pp 38–49

Vassallo C, Proksch S, Zemp T, Gall HC (2018) Un-break my build: assisting developers with build repair hints. In: International conference on program comprehension (ICPC). IEEE

Vassallo C, Schermann G, Zampetti F, Romano D, Leitner P, Zaidman A, Di Penta M, Panichella S (2017) A tale of CI build failures: an open source and a financial organization perspective. In: 2017 IEEE international conference on software maintenance and evolution, ICSME 2017, Shanghai, China, September 17-22, 2017. IEEE Computer Society, pp 183–193. https://doi.org/10.1109/ICSME.2017.67

Vassallo C, Zampetti F, Romano D, Beller M, Panichella A, Di Penta M, Zaidman A (2016) Continuous delivery practices in a large financial organization. In: 32nd IEEE international conference on software maintenance and evolution (ICSME), pp 41–50

Wagner S, Jürjens J, Koller C, Trischberger P (2005) Comparing bug finding tools with reviews and tests. In: Proceedings of the 17th IFIP TC6/WG 6.1 international conference on testing of communicating systems, pp 40–55

Wikis (2019) https://help.github.com/en/articles/about-wikis Accessed: 2019-03-10

Zampetti F, Scalabrino S, Oliveto R, Canfora G, Di Penta M (2017) How open source projects use static code analysis tools in continuous integration pipelines. In: Proceedings of the 14th international conference on mining software repositories. IEEE Press, pp 334–344

Zheng J, Williams L, Nagappan N, Snipes W, Hudepohl J, Vouk M (2006) On the value of static analysis for fault detection in software. IEEE Trans Softw Eng (TSE) 32(4):240–253

Carmine Vassallo is a PhD student in Software Engineering and Research Assistant at the University of Zurich (UZH), Switzerland. He received his BSc and MSc in Computer Engineering from the University of Sannio, Italy. Before joining the Software Evolution and Architecture Lab led by Prof. Harald Gall at UZH, he interned at ING Nederland where he investigated the Continuous Delivery practices adopted by DevOps teams. His work focuses on easing the adoption of Continuous Integration and its continuous improvement.

Sebastiano Panichella is a Computer Science Researcher at Zurich University of Applied Science (ZHAW). His main research goal is to conduct industrial research, involving both industrial and academic collaborations, to sustain the Internet of Things (IoT) vision, where future "smart cities" will be characterized by millions of smart systems connected over the internet, controlled by complex embedded software implemented for the cloud. His research is funded by one SNF Grant and one Innosuisse Project. He is authored (or co-authored) around fifty papers appeared in International Conferences and Journals. These research work involved studies with industrial companies and open source projects and received best paper awards. He serves and has served as program committee member of various international conference and as reviewer for various international journals in the fields of software engineering. He is Editorial Board Member of Journal of Software: Evolution and Process, Review Board member of the EMSE and TOSEM, and Lead Guest Editor of special issues at EMSE and IST Journals. He was selected as one of the top-20 Most Active Early Stage Researchers Worldwide in SE. Website: https://spanichella.github.io

Fabio Palomba is a Senior Research Associate at the University of Zurich, Switzerland. He received the European PhD degree in Management and Information Technology from the University of Salerno, Italy, in 2017. His PhD Thesis was the recipient of the 2017 IEEE Computer Society Best PhD Thesis Award (Italy section). His research interests include software maintenance and evolution, empirical software engineering, source code quality, and mining software repositories. He was the recipient of two ACM/SIGSOFT and one IEEE/TCSE Distinguished Paper Awards at ASE'13, ICSE'15, and ICSME'17, respectively, and Best Paper Awards at CSCW'18 and SANER'18. He serves and has served as a program committee member of various international conferences (e.g., MSR, ICPC, ICSME), and as referee for various international journals (e.g., TSE, EMSE, JSS) in the field of software engineering. Since 2016 he is Review Board Member of EMSE and, since 2019, Editorial Board Member of JSS and Social Media Director of TOSEM. He was the recipient of six Distinguished/Outstanding Reviewer Awards for his reviewing activities conducted for EMSE, IST, JSS, and ICPC between 2015 and 2019.

Sebastian Proksch is a post-doctoral researcher at the University of Zurich, Switzerland, in the group of Prof. Harald Gall. His current research is focused on studying collaborative software engineering processes like Continuous Integration and includes work on static analyses, mining software repositories, and human factors in software engineering. The work is driven by the idea to identify challenges and information needs that developers face in their daily work and to build tailored tools that help developers with overcoming these obstacles. He completed his PhD at Technische Universitt Darmstadt, Germany, under the supervision of Prof. Mira Mezini.

Harald C. Gall is professor of Software Engineering in the Department of Informatics at the University of Zurich, Switzerland. His research interests are in evidence-based software engineering with a focus on quality in software products and processes. This includes long-term software evolution, software architectures, software quality analysis, data mining of software repositories, cloud-based software development, and empirical software engineering. https://www.ifi.uzh.ch/en/seal/people/gall.html

Andy Zaidman is an associate professor at the Delft University of Technology, The Netherlands. He obtained his MSc (2002) and PhD (2006) in Computer Science from the University of Antwerp, Belgium. His main research interests are software evolution, program comprehension, mining software repositories and software testing. He is an active member of the research community and involved in the organization of numerous conferences (WCRE08, WCRE09, VISSOFT14 and MSR18). He is on the editorial board of JSS and EMSE. In 2013 he was the laureate of a Vidi career grant from the Dutch science foundation NWO, while in 2019 he won the Vici career grant, the most prestigious career grant from the Dutch science foundation NWO.

Affiliations

Carmine Vassallo[1] · Sebastiano Panichella[2] · Fabio Palomba[1] · Sebastian Proksch[1] · Harald C. Gall[1] · Andy Zaidman[3]

Sebastiano Panichella
panc@zhaw.ch

Fabio Palomba
palomba@ifi.uzh.ch

Sebastian Proksch
proksch@ifi.uzh.ch

Harald C. Gall
gall@ifi.uzh.ch

Andy Zaidman
a.e.zaidman@tudelft.nl

1 University of Zurich, Zurich, Switzerland

2 Zurich University of Applied Science, Zurich, Switzerland

3 Delft University of Technology, Delft, The Netherlands