Context Is King: The Developer Perspective on the Usage of Static Analysis Tools

Vassalo, Carmine; Panichella, Sebastiano; Palomba, Fabio; Proksch, Sebastian; Zaidman, Andy; Gall, Harald C.

DOI
10.1109/SANER.2018.8330195

Publication date
2018

Document Version
Final published version

Published in
Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER),

Citation (APA)
Vassalo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., & Gall, H. C. (2018). Context Is King: The Developer Perspective on the Usage of Static Analysis Tools. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER),* (pp. 38-49). IEEE . https://doi.org/10.1109/SANER.2018.8330195

Context Is King: The Developer Perspective on the Usage of Static Analysis Tools

Carmine Vassallo*, Sebastiano Panichella*, Fabio Palomba* †,
Sebastian Proksch*, Andy Zaidman†, Harald C. Gall*
*University of Zurich, Switzerland, †Delft University of Technology, The Netherlands

***Abstract*—Automatic static analysis tools (ASATs) are tools that support automatic code quality evaluation of software systems with the aim of (i) avoiding and/or removing bugs and (ii) spotting design issues. Hindering their wide-spread acceptance are their (i) high false positive rates and (ii) low comprehensibility of the generated warnings. Researchers and ASATs vendors have proposed solutions to prioritize such warnings with the aim of guiding developers toward the most severe ones. However, none of the proposed solutions considers the *development context* in which an ASAT is being used to further improve the selection of relevant warnings. To shed light on the impact of such contexts on the warnings configuration, usage and adopted prioritization strategies, we surveyed 42 developers (69% in industry and 31% in open source projects) and interviewed 11 industrial experts that integrate ASATs in their workflow. While we can confirm previous findings on the reluctance of developers to configure ASATs, our study highlights that (i) 71% of developers do pay attention to different warning categories depending on the development context, and (ii) 63% of our respondents rely on specific factors (*e.g.*, team policies and composition) when prioritizing warnings to fix during their programming. Our results clearly indicate ways to better assist developers by improving existing warning selection and prioritization strategies.**

***Index Terms*—Static Analysis, Development Context, Continuous Integration, Code Review, Empirical Study**

I. Introduction

Developers face many challenges in their daily work on evolving software systems [26]. Their job is not only very creative; at the same time, developers also need to avoid, for instance, bugs and security issues. The ever-increasing complexity of source code and constant change make this hard [26]. A means for improving the quality of source code and to reduce bugs are code reviews, in which other developers review changes [13]. However, it does not matter if developers validate their changes themselves or rely on the feedback of others through code reviews, both cases require *human inspection*. This human component introduces considerable manual effort and is also very error-prone, because identifying the bad cases in a pool of changes requires programming experience. Hard and tedious tasks, such as source code analysis, quality assessment, and debugging [22], provide an excellent opportunity for tool support that does not only make life of developers easier, but that also improves the quality of the result. Such *Automatic Static Analysis Tools* (ASATs), i.e., tools that analyze code without executing it, can be integrated in the development process.

Over the years, many such ASATs have been proposed. They can automatically *check code style* [23], *support formal verification* [11], detect *bugs and vulnerabilities* [20], [21], or alert about more general *actionable warnings* [18], [37]. Previous work has shown that ASATs can help in detecting software defects faster and cheaper than human inspection or testing would [22], [5]. Nowadays, ASATs are regularly being used in both open source [4] and industrial [39], [45] projects. The domain has already matured enough that also industrial grade tools exist by now [12].

Despite their advantages, people struggle in proficiently using ASATs due to (i) the high rate of false positives (*i.e.*, alerts that are not actual issues), (ii) low understandability of the alerts, and (iii) lack of effectively implemented quick fixes [22]. So far, related work was mainly concerned with improving the selection strategies to better prioritize the warnings. However, previous studies have shown that there is no golden bullet. Indeed, Zampetti *et al.* [47] found that ASATs related build failures are mainly caused by coding standard violations. In contrast, the most frequently fixed ASAT warnings during code review are related to *coding structure* (e.g., imports, regular expressions, and type resolution) [31]. These results suggest that developers tend to fix different warnings in different stages. We argue that the development contexts (or stages) of usage of ASATs is a strong factor to consider to further improve the prioritization of warnings, which can allow the filtering of the irrelevant ones for the current development activity or context. For this reason, in this paper we analyze the following 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?*

We have conducted two studies to answer these questions. We have first explored the adoption of ASATs in practice through a survey. The survey has involved 42 developers (69% working in industry and 31% open source contributors) that integrate ASATs in their software release pipeline. We have then enforced our findings through semi-structured interviews with 11 industrial developers.

In our studies, we could validate that the prevalent development contexts in which our participants use ASATs are *continuous integration*, *code review*, and *local programming*. In contrast to existing work that proposed *global* solutions for

978-1-5386-4969-5/18/$31.00

better prioritization of warnings [24], [38], we observed in our study that both the selection of ASATs as well as the reaction to specific kinds of warnings, depends on the different development activities and thus, development contexts in which the tools are being applied. While developers say that they *do not configure* ASATs differently in the emerged contexts, the current context actually has an impact on the categories of warnings that are being considered by the developer. This main finding represents the starting point for a more context-based configuration of ASATs, guiding future research in the area of automated warnings configuration and prioritization.

In summary, the contributions of this paper are as follows:

- We present a survey with 42 participants to explore the practical usage of ASATs.
- We have conducted semi-structured interviews with 11 participants to validate our findings from the questionaire.
- We are the first to show the concrete value of considering the *development context* in ASATs.
- We provide insights and potential implications for both ASATs vendors and researchers interested in improving techniques for the automated configuration and warning prioritization of ASATs.

II. Overview of the Research Methodology

Originating from the agile coding movement, modern software development processes are typically structured around three 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 the point-of-view 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 [31]. The typical workflow in *continuous integration* is different. Committed source code is automatically compiled, tested, and analyzed [6], [19]. ASATs are typically used in the analysis stage to assess whether the new software version follows predefined quality standards [47].

In this paper, we conjecture that the development context plays an important role in the adoption and configuration of ASATs, and on the way that the actionable warnings are selected. Figure 1 shows an overview over our methodology that we have used to analyze our conjecture. We have conducted two studies to analyze the impact of the development context and apply two empirical activities, a survey and semi-structured interviews, to generate the required data. We discuss the details of both activities in the following.

A. Survey of ASAT Usage in Open Source and Industry

To explore the usage of ASATs in open-source and industrial projects, we designed a questionnaire. Our survey

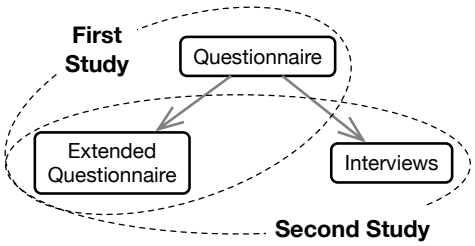

Fig. 1. Overview of the Research Methodology

was implemented using *Google Forms*[1]. In a first step, we relied on advertising the study on social media channels to acquire study participants. To address more participants outside academia, we also applied opportunistic sampling [15] to find open source contributors (OSS) that adopt ASATs in their development process. To this end, we analyzed OSS projects in the TravisTorrent dataset [7] that integrate ASATs in their development process. We could identify these projects from their corresponding source-code repositories by extracting ASATs related options from their configuration files. To avoid sending unsolicited mass emails, we only asked a random sample of 52 developers for their participation.

The survey was available for three months to maximize the amount of collected answers. In the course of this work, we realized that we needed to ask additional questions to make sense of the data, so we extended the initial set of questions of the survey. We kept the original questions untouched and continued collecting data. For this reason, the initial questions were answered by all participants, but the extended questionnaire was only answered by the later participants.

In total we received 44 responses but we had to discard 2 of them because the corresponding respondents declared that they do not use ASATs. 17 of the remaining 42 participants that completed our survey were developers from our list of personal contacts, achieving a return rate of 33%. We announced the extended version of the survey only over the same social media channels as the first part and we found 25 additional participants that answered these further questions. For the latter group, we cannot establish the return rate.

Table I lists demographic information about our survey participants. We had 29 (69%) industrial and 13 (31%) 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" (49%) or "excellent" (36%) developers. Furthermore, 79% of them have more than 5 years development experience, and 43% even more than 10.

We were also interested in profiling the tools our participants use during development. Maven (38%) and Gradle (29%) are the (CI) build tools most commonly used by our participants. However, some participants rely on build tools

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

TABLE I
DEMOGRAPHIC INFORMATION ABOUT OUR SURVEY PARTICIPANTS

Team Size		Projects Size [LoC]	
1-5	24%	1,000-300,000	57%
5-10	37%	300,000-1,000,000	20%
10-15	18%	>1,000,000	5%
>15	21%		
Experience (Years)		**Experience (Rate)**	
1-5	21%	Poor	0%
5-10	36%	Fair	0%
>10	43%	Good	15%
		Very Good	49%
		Excellent	36%

TABLE II
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

like SBT (4%), that is mostly used in Scala development, or Bundler (2%), the most common 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 [16], [17]. The largest part of our respondents declare to be supported by distributed version control systems as GitHub (25%), Gitlab (19%) or Bitbucket (9%) during the code review process. Nevertheless, some participants still tend to rely on a dedicated code review tool, *i.e.*, Gerrit (19%), or to use an informal process (16%).

B. Semi-Structured Interviews with Professional Developers

We have interviewed industrial experts that use ASATs daily. This has helped us to overcome the typical limitations of a survey, e.g., the lack of conscientious responses. The interviews complement the survey. They provide another perspective on the previous results and can possible explain observations from the questionnaire.

We have defined a guideline for the interviews, but decided to adopt a semi-structured interview format [36] 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 II, they cover different domains. 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 have participants from both perspectives: (i) developers that are actually using 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 (40%), a result that is in line with previous work conducted in industry [44]. The other ASATs that are most-employed in our participants' companies are Findbugs (13.6%), Checkstyle (9.1%) and IDE plugins, e.g., CodePro (9.1%).

C. Data Analysis

We have used the data of both the survey and the interviews to conduct two studies that are highlighted in Figure 1 with the dotted circles[2]. The goal of the *first study* was to (i) assess the contexts in which developers use ASATs and (ii) understand whether they modify ASATs configuration. We have used the initial set of questions that were answered by all survey participants (see Section III).

The goal of the *second study* was to understand how the context influences their selections of warnings to which they react. This study is based on the extended questionnaire and on the semi-structured interviews (see Section IV). The overall findings of this work are then discussed in Section V.

III. THE DEVELOPMENT CONTEXTS INTEGRATING ASATS

The *goal* of this preliminary study is to understand (i) what the development contexts are in which developers adopt ASATs and (ii) how they configure them in the various contexts, 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 of our survey (more details about them in the next sub-sections) and (ii) *objects*, that are the specific ASATs used by our respondents.

A. Survey Design

Our initial questionnaire 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. We have grouped our various questions in Table III into three topics: (i) Background, (ii) Adoption of ASATs, and (iii) Configuration of ASATs.

[2]The surveys' responses, relevant statements form interviews and further data analyses can be found at http://www.ifi.uzh.ch/seal/people/vassallo/VassalloSANER18.zip.

TABLE III
SURVEY QUESTIONS. (MC: MULTIPLE CHOICE, S: STATEMENTS, C: CHECKBOXES, O: OPEN ANSWER)

Section	ID	Summarized Question	Type	# Resp.
Adoption	Q1.1	To what extent do you use ASATs during your activities?	MC	42
	Q1.2	During which activities do you use ASATs?	O	34
	Q1.3	Which ASATs do you usually work with?	C	41
	Q1.4	If you use more than one ASAT, why you're adopting more than one ASAT and in which context?	O	24
	Q1.5	In which step of software development do you usually rely on the suggestions provided by ASATs?	C	41
Configuration	Q2.1	To what extent do you change configuration of ASATs?	MC	41
	Q2.2	Do you use different configurations when working (i) in CI, (ii) Code Review, (iii) locally? If so, why?	O	28
	Q2.3	While configuring, do you pay attention to different warnings (i) in CI, (ii) Code Review, (iii) locally?	O	11
	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	23
	Q2.5	To what extent do you integrate warnings suggested by ASATs during CI?	MC	40
	Q2.6	To what extent do you integrate warnings suggested by ASATs during Code Review?	MC	38
	Q2.7	To what extent do you integrate warnings suggested by ASATs locally?	MC	36

The BACKGROUND questions provided us with the demographic information that we have reported in Section II. 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 ASAT usage in practice. Specifically, the ADOPTION OF ASATS 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 them 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 used multiple ASATs (Q1.4) and in which development contexts (Q1.5).

The CONFIGURATION OF ASATS section (Q2.1-Q2.7) was 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.* [4]). For this reason, we asked our participants when and which are the contexts where they change the configuration of ASATs. Then we asked them how frequently they fix warnings suggested by ASATs in the different considered contexts.

B. Adoption of ASATs

Most of the respondents (38%) declared to use ASATs multiple times per day, while 31% use them on average once per day. As shown in Figure 2 the most used ASATs are Findbugs (19%), Checkstyle (18%) and PMD (14%). Then, SonarQube and ESLint are preferred respectively by 11% and 7% of our respondents. Few participants mention other tools, *e.g.*, Pylint, JSHint, Flake8, Checkmarx. The participants who regularly use ASATs (*i.e.*, multiple times per day, or once per day) also indicated the development activities during which they usually adopt the tools (Q1.2). This information allows us to answer RQ_1.

To verify the contexts in which developers use ASATs, two of the authors (*sorters*) performed a closed card sorting [40] of

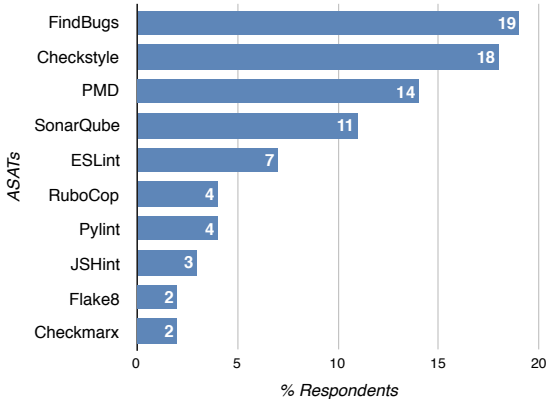

Fig. 2. Top-10 ASATs used by our participants.

the described development activities where the participants use the ASATs tools. Thus to compute the development contexts in which developers use ASATs we followed three steps.

1) The two sorters assigned independently each *development activity* provided by the participants, (*i.e.*, the cards) to one 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).
2) We computed Krippendorff's alpha [25] to determine the interrater reliability of the results of the first independent card sorting.
3) We involved a third author 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.

The results of card sorting are shown in Table IV. 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 5, and they never completely disagreed. We computed

TABLE IV
DEVELOMPENT ACTIVIES WHERE ASATS ARE INTEGRATED.

Activity Name	# Resp.	Development Context			Agreement
		LP	CR	CI	
Code Maintenance	4	✓	✓	✓	Full
Code Reviewing	17		✓		Full
CI Build	8			✓	Full
In-Editor typing	1	✓			Full
Pre-commit	2	✓	✓		Partially
Pre-push	2	✓			Full
Build cycle	1			✓	Full
Refactoring	4	✓	✓		Partially
Jenkins stage	1			✓	Full
Debugging	2	✓			Partially
Documentation	1	✓			Partially
Quality Check	1	✓	✓	✓	Full
In-IDE Typing	1	✓			Full

Krippendorffs alpha coefficient to assess the reliability of the performed sorting. With a score of 0.68, it shows an acceptable agreement [25]. 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 II. Moreover, from the results of Q1.5 we found that 37% of our participants rely on them in CI, 33% in CR and 30% in LP.

Finding 1: Developers use ASATs in three different 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 important reason to combine several ASATs seems to be that they *"cover different areas"*, *i.e.*, different rulesets. 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, several 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)".

Based on the answers reported above, we formulated a first hypothesis to be validated:

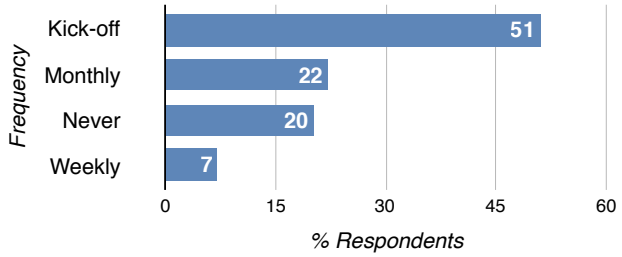

Fig. 3. When ASATs are configured.

Hypothesis 1: Developers intend to enable different warnings in different contexts.

C. Configuration of ASATs

Beller *et al.* [4] 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 such analysis are shown in Figure 3. The general findings by Beller *et al.* [4] are confirmed: indeed, in more than half of the participants (51%) 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 (22%).

To better investigate the motivations behind the update of 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 (75%) 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.

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

Despite this general trend, a considerable portion (25%) 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 25% 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 75% of our respondents), some developers might need or want to configure ASATs differently depending on the development context. Thus, we defined a second hypothesis:

Hypothesis 2: Despite their tendency to not configure ASATs, developers pay attention to different types of warnings depending on the context in which ASATs are integrated.

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.

IV. The Impact of Development Contexts in the ASATs Configuration

From the answers the developers provided in the context of RQ_1 we came up with two hypotheses that suggest how *context-aware ASATs* might be useful for developers. The *goal* of this second study is to verify these hypotheses. 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 the next sections, 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.

A. Study Design

1) Extended Questionnaire Design: As described in Section II, we extended our initial survey by including additional questions about Context-Based Usage that are listed in Table V. More specifically, we focused 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.* [10] 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 [24], [37] and from the popular question and answer sites StackOverflow (*e.g.*, [41], [42]) and Reddit (*e.g.*, [34], [35]), which are among the top discussion forums for developers [8]. In the latter case, two of the authors of this paper manually went over the developers' discussions looking for possible indicators expressing the likely motivations pushing developers into using ASATs in different ways.

2) 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 disclose their viewpoints about context based warnings. The guide was split into three sections. In the first section, Background, we asked years of experience, study degree, programming languages used, role in the company together with its size/domain and development contexts where our interviewees adopt ASATs. The second section called Contexts' Understanding was about the development contexts put in place in the organization the participant belonged to. On one hand, we wanted to understand their process to review and build new software. Different to local programming, this process is usually regulated and followed by all developers. On the other hand, we needed to know how they use ASATs. In the last section, Usage of ASATs in each context, we let our interviewees think about the differences in the usage of ASATs in 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.

B. Main Factors Affecting the Warning Selection

Figure 4 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).

TABLE V
ADDED SURVEY QUESTIONS RELATED TO THE CONTEXT-BASED USAGE OF ASATS. (O: OPEN QUESTION, S: STATEMENT)

Section	ID	Summarized Question	Type	# Resp.
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	25
	Q3.2	Which are the warning types that are more likely to be fixed during Continuous Integration?	O	25
	Q3.3	Which are the main factors you consider when deciding the set of warnings to look at during Code Review?	S	25
	Q3.4	Which are the warning types that are more likely to be fixed during Code Review?	S	25
	Q3.5	Which are the main factors you consider when deciding the set of warnings to look while working locally?	S	25
	Q3.6	Which are the warning types that are more likely to be fixed while working locally?"	S	25

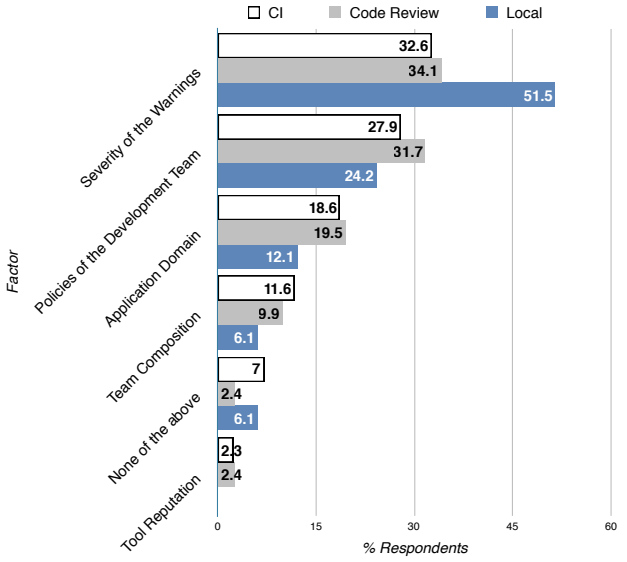

Fig. 4. Main Factors while selecting warnings in different contexts.

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 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 *"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 [22] 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. 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).

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

C. 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

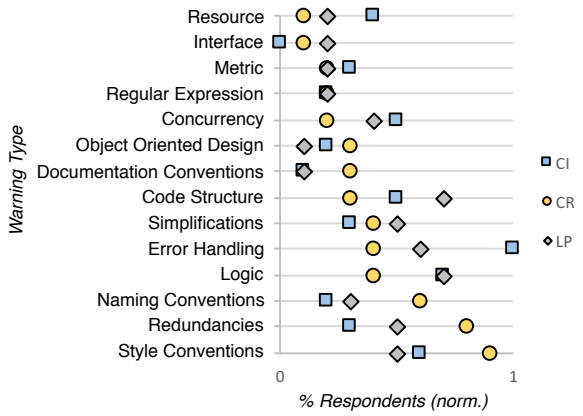

Fig. 5. Normalized Actionability of Different Warning Types

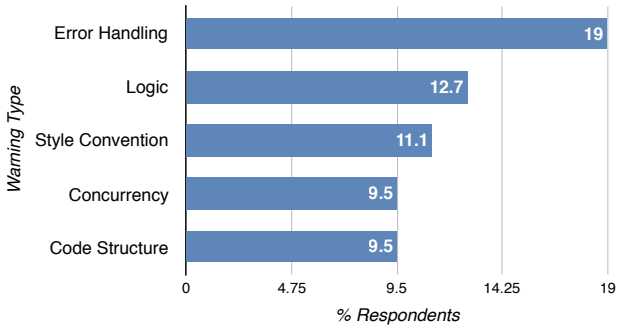

Fig. 6. Top-5 warnings that are likely to be fixed in CI.

independent as possible from specific ASATs, we adopted the *General Defect Classification* (GDC) proposed by Beller *et al.* [4] as the list of warnings types.

Figure 5 illustrates 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* [1] in order to better explain to what extent each warning type is monitored 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 (Figure 6), Code Review (Figure 7) and Local Programming (Figure 8). 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 (third most selected in Figure 6) and locally (fourth in Figure 8), but specifically during code review: it is the warning type selected by the majority of our respondents, as shown in Figure 7. This result confirms findings of previous work [3], [31] 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 the code understandability. S9 and S10 confirm during the interviews that style-related issues are crucial points to address during

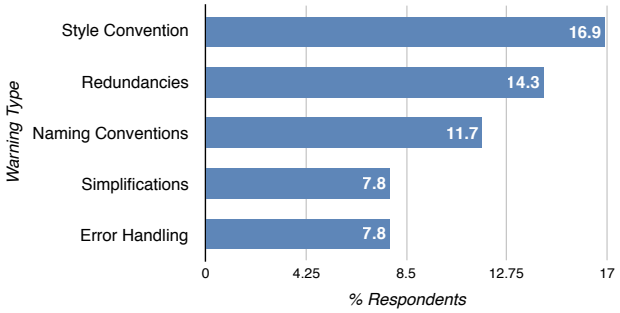

Fig. 7. Top-5 warnings that are likely to be fixed in Code Review.

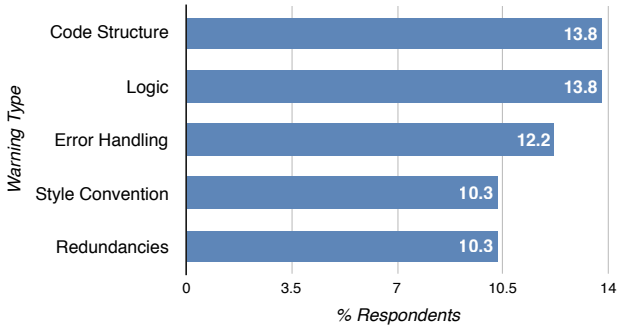

Fig. 8. Top-5 warnings that are likely to be fixed locally.

code review. Furthermore, S9 considered it also very valuable while working locally.

The importance of *Simplifications* differs along the contexts as shown in Figure 5. Warnings in this category highlight code that could be simplified to improve readability and understandability. It is only among the top-5 warnings for code review (Figure 7), confirming again previous findings in the field [3]. However, overall, we observe that respondents mostly select this type during local programming, where S9 states to use ASATs to make the code easier to comprehend.

Redundancies concern redundant pieces of code or artifacts that can be safely removed. It is perceived as a very important issue during code review (the second among the most selected warnings) but also locally although if in a lower extent. Nevertheless it is sometimes selected also during CI as S1 usually does.

Our respondents also pointed out that they mainly look at *Naming Conventions* during code reviews (third most selected warnings in Figure 7), while the *Logic* warnings that are concerned with comparisons, control flow and algorithms are widespread in CI and local programming. Indeed, they received the same number of votes in Figure 5. Thus, this result highlights that developers have different goals in different contexts, thus leading to a different usage in each of them.

Error Handling is the most selected warning in CI, *i.e.*, occupies the first position among the chosen warnings. It is quite popular locally (the third most voted in Figure 8) and less important but still in the top-5 in code review. Indeed only S1 and S3 monitor this warning type during code reviews, while mostly relying on the CI server to spot such issues.

Concurrency refers to defects that appear while sharing resources among multiple interactive users or application programs at the same time is the fourth warning type selected in CI (Figure 6), meaning that developers are interested in performing such checks that are usually time consuming by enacting a new build.

Code Structure shares the first position with *Logic* in the warnings that are likely to be fixed locally (Figure 8). 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 very important for our respondents while working locally. The same percentage of respondents indicate the errors pertaining to program *Logic* as warning type usually selected locally.

Finding 4: Developers consider important different warning types in different contexts. When programming in the IDE, they observe warnings related to code structure and logic; when performing code reviews they mainly look at style conventions and redundancies; during CI, they watch handling errors and homogenize code logic and concurrency.

V. Discussion

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

For the RQ1 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 along the contexts: 37% of our survey participants rely on ASATs while integrating code changes in an existing project, 33% while reviewing code and 30% while working locally.

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

In RQ2, we discovered that 51% 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.* [4], who showed that developers did not change ASATs configuration often. Despite its usage in these three different contexts, the majority of developers (75% 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.

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.

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

In RQ3, 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.

The actual ASATs' configurations do not reflect the developers' perception of warnings to monitor in each development context.

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

Biased Perception: We have seen a contrast between what developers think about ASATs' configuration and what they pay attention to in practice. This suggests the need for future research of novel techniques that can estimate the actual factors that influence the warnings selection, *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 impose the adoption of specific warnings to developers. These results suggest the need of future research devoted to the implementation of novel tools that are able to estimate goods weights for the context specific warning selection of ASATs. To this end, telemetry data about developer activities (e.g., [32], [9]) might provide useful input for personalized ASATs 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 proper support the adoption of ASATs in each context.

VI. 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 firstly shared the surveys using an online survey platform and forced participants to fill the main questions. Secondly, we complemented the questionnaires by involving 11 industrial experts that use ASATs on a daily basis. We plan to conduct a mining software repository study to confirm the current qualitative findings in our future work.

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 [30] to collect perceived relevance of some ASATs practices. To limit random answers, we provided to the participants 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 [40] 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 a solid development experience. Clearly, it is possible that some of our results partially generalize to other organizations and open source companies.

VII. Related Work

In past and recent years, ASATs have captured the attention of researchers under different perspectives. Flanagan *et al.* [14] investigated the usefulness of two ASATs, *i.e.*, ESC-Java and CodeSonar, discovering that they have reliable performance. Wagner *et al.* [46] 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 depend on the developers programming style. At the same time, Ayewah *et al.* [2] showed that the defects reported by FindBugs are issues that developers are actually interested in to fix. Zheng *et al.* [48] 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.* [33] 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 *et al.* [27] found that the warning density of static analysis tools is correlated with pre-release defect density.

Kim and Ernst [24] 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. In addition, Thung *et al.* [43] and Nanda *et al.* [28] 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 tough a substantial proportion of them is still not captured.

Beller *et al.* [4] 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.* [47] and Panichella *et al.* [31] 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*. Nurolahzade *et al.* [29] 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.* [31] and Zampetti *et al.* [47] 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.

VIII. Conclusion

This paper has presented the developers' perspective on the usage of Automatic Static Analysis Tools (ASATs) in practice. We have conducted two studies among developers working in industry or contributing to open source projects. We have first explored the adoption of ASATs in practice through a survey and have then enforced our findings in semi-structured interviews with industrial experts. Our findings show 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 configuration when working in different contexts, they assign different priorities to different warnings along the contexts.

Acknowledgments

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. This research was partially supported by the Swiss National Science Foundation through the SNF Projects Nos. 200021-166275 and PP00P2_170529.

REFERENCES

[1] L. Al Shalabi, Z. Shaaban, and B. Kasasbeh. Data mining: A preprocessing engine. *Journal of Computer Science*, 2(9):735–739, 2006.

[2] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In M. Das and D. Grossman, editors, *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*, pages 1–8. ACM, 2007.

[3] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th working conference on mining software repositories*, pages 202–211. ACM, 2014.

[4] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman. 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 - Volume 1*, pages 470–481. IEEE Computer Society, 2016.

[5] M. Beller, G. Gousios, and A. Zaidman. How (much) do developers test? In *37th IEEE/ACM International Conference on Software Engineering (ICSE 2015)*, pages 559–562. IEEE Computer Society, 2015.

[6] M. Beller, G. Gousios, and A. Zaidman. Oops, my tests broke the build: an explorative analysis of travis CI with github. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 356–367. IEEE Computer Society, 2017.

[7] M. Beller, G. Gousios, and A. Zaidman. TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration. In *Proceedings of the 14th working conference on mining software repositories*, 2017.

[8] CryptLife. Top ten forums for programmers. https://www.cryptlife.com/designing/programming/10-best-active-forums-for-programmers, 2017.

[9] M. Dias, D. Cassou, and S. Ducasse. Representing Code History with Development Environment Events. In *International Workshop on Smalltalk Technologies*, 2013.

[10] D. A. Dillman, J. D. Smyth, and L. M. Christian. *Internet, phone, mail, and mixed-mode surveys: the tailored design method*. John Wiley & Sons, 2014.

[11] V. D'silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[12] P. Emanuelsson and U. Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21, 2008.

[13] M. E. Fagan. Advances in software inspections. *IEEE Trans. Software Eng.*, 12(7):744–751, 1986.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

[15] L. Gibbs, M. Kealy, K. Willis, J. Green, N. Welch, and J. Daly. What have sampling and data collection got to do with good qualitative research? *Australian and New Zealand journal of public health*, 31(6):540–544, 2007.

[16] G. Gousios, M. Pinzger, and A. van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 345–355, New York, NY, USA, 2014. ACM.

[17] G. Gousios, A. Zaidman, M.-A. D. Storey, and A. van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *37th IEEE/ACM International Conference on Software Engineering (ICSE 2015*, pages 358–368. IEEE Computer Society, 2015.

[18] S. Heckman and L. Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 53(4):363–387, 2011.

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

[20] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[21] C. Inc. Effective management of static analysis vulnerabilities and defects. 2009.

[22] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge. Why don't software developers use static analysis tools to find bugs? In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 672–681. IEEE Computer Society, 2013.

[23] S. C. Johnson. *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977.

[24] S. Kim and M. D. Ernst. 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, pages 45–54. ACM, 2007.

[25] K. Krippendorff. *Content analysis: An introduction to its methodology*. Sage, London, 2nd edition, 2004.

[26] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1980.

[27] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 580–586, 2005.

[28] M. G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran. Making defect-finding tools work for you. In *Proceedings of the International Conference on Software Engineering (ASE) - Volume 2*, pages 99–108, 2010.

[29] M. Nurolahzade, S. M. Nasehi, S. H. Khandkar, and S. Rawal. 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*, pages 9–18, 2009.

[30] B. Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter Publishers, 1992.

[31] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol. 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*, pages 161–170, 2015.

[32] S. Proksch, S. Nadi, S. Amann, and M. Mezini. Enriching in-ide process information with fine-grained source code history. In *International Conference on Software Analysis, Evolution, and Reengineering*, 2017.

[33] F. Rahman, S. Khatri, E. T. Barr, and P. T. Devanbu. Comparing static bug finders and statistical prediction. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 424–434, 2014.

[34] Reddit. Php static analysis tools. https://www.reddit.com/r/PHP/comments/5d4ptt/static_code_analysis_tools_veracode/, 2017.

[35] Reddit. Static analysis tools. https://www.reddit.com/r/programming/comments/3087rz/static_code_analysis/, 2017.

[36] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164, Apr. 2009.

[37] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proceedings of the 30th international conference on Software engineering*, pages 341–350. ACM, 2008.

[38] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. G. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 341–350, 2008.

[39] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In A. Bertolino, G. Canfora, and S. G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 598–608. IEEE Computer Society, 2015.

[40] D. Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[41] StackOverflow. Static analysis tool customatization. https://stackoverflow.com/questions/2825261/static-analysis-tool-customization-for-any-language, 2017.

[42] StackOverflow. Static analysis tools. https://stackoverflow.com/questions/22617713/whats-the-current-state-of-static-analysis-tools-for-scala, 2017.

[43] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu. 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)*, pages 50–59, 2012.

[44] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. D. Penta, and S. Panichella. 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*, pages 183–193. IEEE Computer Society, 2017.

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

[46] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger. 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*, pages 40–55, 2005.

[47] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 334–344. IEEE Press, 2017.

[48] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering (TSE)*, 32(4):240–253, 2006.