Why Do Software Developers Use Static Analysis Tools?

A User-Centered Study of Developer Needs and Motivations

Lisa Nguyen Quang Do, James R. Wright, and Karim Ali

Abstract—As increasingly complex software is developed every day, a growing number of companies use static analysis tools to reason about program properties ranging from simple coding style rules to more advanced software bugs, to multi-tier security vulnerabilities. While increasingly complex analyses are created, developer support must also be updated to ensure that the tools are used to their best potential. Past research in the usability of static analysis tools has primarily focused on usability issues encountered by software developers, and the causes of those issues in analysis tools. In this article, we adopt a more user-centered approach, and aim at understanding why software developers use analysis tools, which decisions they make when using those tools, what they look for when making those decisions, and the motivation behind their strategies. This approach allows us to derive new tool requirements that closely support software developers (e.g., systems for recommending warnings to fix that take developer knowledge into account), and also open novel avenues for further static-analysis research such as collaborative user interfaces for analysis warnings.

Index Terms—Program analysis, Development tools, Integrated environments, Graphical environments, Usability

1 Introduction

From simple linters such as Checkstyle [1] to more complex tools such as CodeSonar [2] and Fortify [3], static analysis has improved over time to detect more complex bugs faster and with better accuracy. While the underlying analyses grow more efficient, the usability of the tools improves at a slower rate. Previous studies have reported on recurrent usability problems such as poor explainability and slow updates [4], [5], [6], [7]. Such usability issues may lead to misinterpreted warnings and tool abandonment.

While past studies focus on analysis correctness and usability issues reported by tool users, we offer a new perspective on the usage of static analysis tools in practice, by focusing on the usage context of the tools, and the developer motivations and strategies when working with them. To understand how to best support developers when designing or setting up static analysis tools in industry, we apply the principles of user-centered design [8], [9] to study the usage context of the tools from the developer's perspective. This approach helps us understand the goals that developers have when using analysis tools, what they expect from the tools when working with them, and what can be done to better support them. We then derive requirements and applicable guidelines for designing and using analysis tools.

We present three categories of findings: (1) findings that confirm results of prior work (e.g., soundness issues in analysis tools), (2) findings that contrast prior work (e.g., developers are not as interested in style warnings as performance warnings), and (3) findings that highlight novel research areas for static analysis (e.g., tool support for collaborative interfaces).

To understand the usage context of analysis tools in practice, we conducted a survey in industry in collaboration with Software AG [10], an international software vendor based in Germany, and present in more than 70 countries. It is active in areas such as database management systems, big data analytics, business analytics, networking, software development, data transfer, and cloud solutions. As a large software vendor, Software AG has a strong interest in the functionality and security of their software products. Among other security measures, they use a large array of static analysis tools, which are the focus of our survey.

Our survey focuses on the usage of 17 analysis tools at Software AG, with 87 of its software developers. We report on the developers' goals, motivations, and strategies when they use analysis tools, how those three aspects influence the way they interact with the tools, and which tool features could thus support them best. In addition, we illustrate some of the findings of our survey through a study of the analysis results of Checkmarx [11], a major analysis tool used by Software AG, on two large projects of the company.

In this article, we make the following contributions:

- We present the results of a survey over 87 developers and 17 analysis tools, and a deeper study of Checkmarx warnings, focusing on how and why developers integrate static analysis in their work.
- We report on how analysis tools are integrated in Software AG's development environment, the different types of tools that are made available to the developers, and what the developers would prefer using.
- We describe how developers use analysis tools in their daily work, what their goals are when opening an analysis tool, and if those goals are met when they close it.
- We present the strategies used by developers when they perform different tasks with analysis tools: warning prioritization, determining whether a warning is a false positive or a true report, understanding a warning, and how they handle warnings they do not understand.
- In light of developer motivations, strategies, and goals, we identify recommendations and new research areas for designing and using analysis tools in industry, namely

- *L. Nguyen Quang Do was with Paderborn University, Germany (now with Google, Switzerland).*
  E-mail: lisa.nguyen@upb.de
- *J. Wright and K. Ali are with the University of Alberta, Canada.*
  E-mails: firstname.lastname@ualberta.ca

building workflows that take the developer work time into account, integrating developer heuristics in the tools to encourage or discourage certain behaviour, and building collaborative platforms to leverage the knowledge of developer teams.

2 Related Work

We present related work on the use of static analysis tools, and past studies reporting on their user-experience issues.

2.1 Usability of Static Analysis Tools in Practice

Static analysis has been used in industry since the 1970s. At first aimed at compiler optimization, its uses are now extended to bug and security vulnerability spotters, or coding automation such as code completion or refactoring [12]. The first large static analysis tools dedicated to bug and vulnerability detection appeared in the early 2000s, such as Coverity [13] or Fortify [3]. Since then, the adoption of static analysis tools in industry has steadily increased to detect bugs as early as possible in the software development process, thus cutting fixing costs by multiple factors [14]. In more recent years, applications of static analysis tools have expanded, supporting more languages and use cases, and detecting a growing set of bugs and vulnerabilities. From lightweight checkers run in the Integrated Development Environment (IDE) such as FindBugs [15] to more complex analyses such as Checkmarx [11] which are typically run during nightly builds, open-source and commercial tools are used and recommended by security authorities such as OWASP [16] and CERN [17].

Despite its success, static analysis has been known for specific user-experience issues since its first applications in industry. Vorobyov et al. [18] compare model checking and static program analysis by focusing on the precision of both approaches, and discuss the causes of their limitations in the approach's functionalities (e.g., handling internal libraries). Bessey et al. [4] report on the experience of software developers with Coverity since its release in industry in 2002, showing that bad warning explainability, unclear tool configurations, and a high number of false positives have been issues with static analysis tools early on. Christakis et al. [19] survey developers and study live site incidents at Microsoft to extract pain points (e.g., false positives and bad warning messages) and potential improvements to the analyses (e.g., sources of unsoundness and reporting locations).

Those studies primarily focus on the tools and their usability issues, deriving features for better developer support with respect to the analysis (e.g., improving precision). In contrast, our focus is not the tools themselves. Instead, we approach the problem from the developers' perspective, and report on their motivation for using the tools, their strategies for using the tools, and, as a result, derive a different set of supporting features (e.g., collaboration features).

2.2 Developer Motivation and Behavior

The two studies closest to our own also report on developer motivations and how they influence the requirements for static analysis tools. Layman et al. [20] study the strategies of 18 student developers using the analysis tool AWARE. They extract requirements that differ from other studies such as the need for integrating the user's perception of severity in the severity rating of a warning. Johnson et al. [6] interview 20 developers on their experience with various analysis tools, focusing on the usability issues encountered in those tools and why they occur, from the point of view of the developer. This approach enabled the authors to present different requirements such as the need for better explanations in warning descriptions.

We build on both studies and conduct ours on a company-wide scale. Instead of reporting on usability issues, we focus on the usage context, including company policies and developer schedules, and how it affects the way that developers work with analysis tools. We then derive requirements for building and using analysis tools.

Vassallo et al. [21] survey 42 developers and interview 11 experts who install and configure static analysis tools in industry. They focus on the usage context of the tools when developers prioritize warnings to fix first. They identify factors that developers take into account when making the decision, and show that certain coding contexts lead to different decisions. While we do not go in the detail of how particular contexts influence the developer's actions, our study also reports on developer strategies for prioritizing warnings and extends to other developers actions—such as distinguishing true from false positives or asking colleagues for help— and to different aspects of analysis tool usage (e.g., developer constraints and motivations).

Ayewah et al. [7] report on developer usage of FindBugs, in particular the importance of warning severity in selecting which warnings to fix first, and how developers handle false positives. In a follow-up study, Ayewah et al. [22] observe student developers while using FindBugs to extract the factors that impact warning understandability. The findings of both studies are consistent with our survey, but we further investigate developer motivations and reasons for which they make those choices. Additionally, our study covers more aspects of how developers work with analysis tools such as how they decide which warnings are false positives, and what they do with warnings they do not understand.

Lewis et al. [5] compare different analyses and observe which one enables Google developers to be more efficient. Factors such as a bias towards new warnings or actionable messages were shown to be factors of interest to the developers. Similar to Ayewah et al. [22], the authors conduct their study in a controlled environment, not accounting for realistic usage contexts such as time constraints and real-life motivations, which we highlight in our study.

Zampetti et al. [23] study the usage of static analysis tools in open-source projects. Overlapping with our study, they report on the different types of issues that are reported and how they are addressed. Our study covers the additional human aspect of tool usage: how and why developers interact with such tools.

Zheng et al. [24] study the usage of static analysis tools at Nortel Networks. They focus on the economical aspect of static analysis tool usage, and —among other results— report on the types of warnings their tools report, which is the only intersection with our study.

3 Study

To understand how developers interact with static analysis tools, we conducted a two-part study. First, we sent a survey across the main development teams at Software AG, asking developers about their experience with static analysis tools. In a second part, we were given access to the reports of analysis runs with Checkmarx [11], one of the major analysis tools used by Software AG. Checkmarx is a static analysis tool that supports 20 programming languages, and that can be used as a standalone tool with a web interface or in different Integrated Development Environments as a plugin. It is part of the continuous integration system at Software AG, and is used by a large variety of projects to detect software bugs and security vulnerabilities. Its interfaces provide management overviews of the projects' health, and detailed information about individual warnings, which developers use to communicate about the warnings and to fix them. The analysis reports also include information on how developers handled the warnings over several months, for two of Software AG's major projects, which we anonymize as Application 1 and Application 2 at the company's request. We use this data to complement the survey answers.

Our study addresses the following research questions:

RQ1: How are analysis tools integrated in the development environment?
RQ2: In which usage contexts do developers use analysis tools, and with which goals?
RQ3: What are the strategies that developers apply when working with analysis tools?
RQ4: What are the features that analysis tools should provide to developers to support them?

In this section, we present the composition of the survey, the analysis reports, and our methodologies for designing the survey and extracting the data. The complete list of questions and anonymized responses is available online [25].

3.1 Survey Design

To answer **RQ1**–**RQ4**, we designed a survey composed of 40 questions (referred to as **Q1**–**Q40**) grouped into the following six categories. Unless specified otherwise, all questions are multiple choice with an "Others" free-text field.

1) *Participant information:* We asked participants how long they have worked as a developer (**Q1**) and which programming languages they work with (**Q2**).
2) *General use of static analysis tools:* We asked developers general questions on how analysis tools are used at Software AG. This category includes questions on which analysis tools they use at the moment (**Q4**), when those tools are run in the project (**Q5**), who configures them (**Q6**), what kind of issues are detected (**Q7**), what kind of issues they would like the tools to report (**Q8**), if they fix the analysis warnings themselves (**Q9**), and who reviews the fixes (**Q29**).
3) *Reporting warnings:* This category reports on the formats in which the tools report warnings (**Q10**), the formats the developers would prefer (**Q11**–**Q12**), how long developers take to fix a warning (**Q13**), and how long they typically wait before their fix is verified (**Q14**). In free-text, we also asked developers to comment on the reporting systems of their analysis tools (**Q15**).

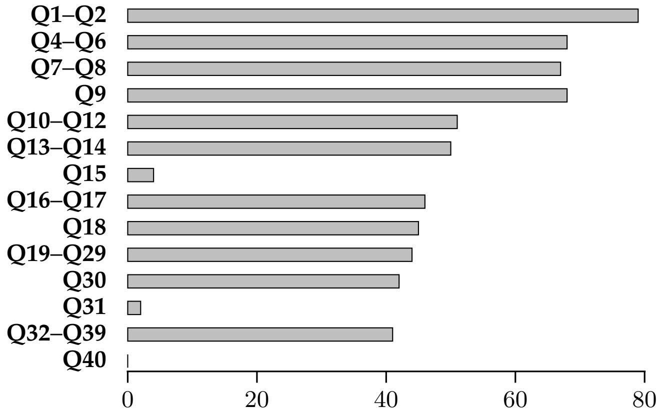


Figure 1: Number of Responses per Groups of Questions.

4) *Working context:* We asked developers which analysis tool they use the most (**Q16**), and focus on that tool for the rest of the category. We asked them how often (**Q18**), when (**Q19**), and where (**Q20**) they use it, how long they use it for (**Q21**–**Q22**), and why they use (and stop using) it (**Q23**–**Q24**). We also queried developers about which parts of the tool's interface they use the most, when they open (**Q26**), work with (**Q28**), and close (**Q27**) the tool, and if they are using the default layout of the interface (**Q25**).
5) *Features of analysis tools:* **Q30** and **Q31** (the latter, as a free-text question) asked developers to evaluate how important different tool features are to them.
6) *Fixing analysis warnings:* This category reports on the ratio of warnings developers investigate (**Q32**), understand (**Q35**), and for which they seek help from colleagues (**Q38**). It details the strategies used by developers to choose which warnings to investigate (**Q33**–**Q34**), the reasons why certain warnings are difficult to understand (**Q36**–**Q37**), and why developers ask for help (**Q39**). Finally, (**Q40**) asks about final comments on analysis tools as a free-text question.

We ran a pilot survey with five developers, after which we compacted the survey so that it could be completed in approximately 20 minutes. We namely removed questions similar to **Q30** and **Q31** that asked developers to which extent their current tools support the features.

3.2 Methodology and Data Extraction

We reached out to 120 developers at Software AG (two thirds of the development force) and received 87 responses, yielding an exceptionally high response rate of 72.5%, covering a fair share of the company's application domains. From those participants, 53 developers completed the survey in full, yielding a drop rate of 39.1%. Figure 1 details the response rates. In this article, when we report on percentages of participants, we take the number of responses to the corresponding question as the baseline, instead of the overall number of 87 participants. The percentages may also add up to more than 100%, because some questions allow the selection of multiple responses. We only report on responses reported by more than one participant.

Our survey gathers information from a diverse group: 46.8% of the participants have 2–5 years of experience as software developers, 25.3% have 5–10 years, 13.9% have 1–2 years, 10.1% have more than 10 years, and 3.8% have less than a year of experience (**Q1**). While the large majority work with Java (91.1%), Javascript (38%), C/C++ (10%), PHP (7.6%), Python (7.6%), and 12 other languages, each used by fewer than 2.5%, are also used (**Q2**). Due to Software AG's policies on the usage of static analysis tools, all participants have experience with them.

All survey questions but three are multiple-choice questions, for which we straightforwardly report the results. We attribute the high response rate to this multiple-choice format which makes it easier and quicker to answer, and to the company's internal publicity of the survey. The survey was designed after a discussion with an experienced Software AG developer, so the suggestions given in the multiple-choice questions covered most of the developers' answers. In most cases, we could re-categorize answers from the "Others" fields in existing categories (e.g., "15" was recategorized in "$>$ 10 years" for **Q1**). We discarded some of them when they clearly did not answer the question (e.g., "Not applicable" for **Q24**). The remaining answers were left in the "Others" category, but were not numerically significant enough to report on. Similarly, we do not report on the free-text questions, which had 3, 2, and 0 answers, respectively.

3.3 Analysis Reports

To complement the survey with respect to developer strategies for triaging and prioritizing analysis warnings (**RQ3**), we analyzed the reports of Checkmarx, one of the main static analysis tools used at Software AG. Checkmarx is deployed on a large number of projects at Software AG, as part of their global effort to improve the quality of their code, and is used by 51.2% of the survey participants.

Checkmarx is a *dedicated tool*, meaning that it is independent of the tools used by developers (e.g., code editors or project management systems). Checkmarx has an elaborate web-based Graphical User Interface (GUI) that provides developers with detailed information such as warning categories (e.g., SQL injection), an estimate of their severity, general warning statistics, etc. Checkmarx also allows developers to comment on the warnings, for example, marking them as false positives or fixed.

We studied the analysis reports of Application 1 which contains 8 sub-projects, varying from 56,000 to 1,550,000 LOC, and Application 2 which has 4 sub-projects, ranging from 270,000 to 6,650,000 LOC. We analyzed scans from spring 2017 to December 2018, except for 6 sub-projects of Application 1, 3 of which have been using Checkmarx since winter 2018, and 3 others, since winter 2017. We use the outcomes of our analysis to support **RQ3** in Section 6.

4 Industrial Deployment of Analysis Tools

In the past few years, Software AG has strived to ensure the quality and security of its software through the use of analysis tools. Software AG continuously encourages its development teams to use static analysis (and other tools) to ensure code quality and security. Individual projects and developers can use their own analysis tools independently, resulting in a variety of analysis tools deployed across the company, going from simple linters that return results almost immediately such as SonarLint to more complex tools such as Fortify, which are slower and typically run as part of nightly builds. In addition, in the past years, global efforts across the company have recently resulted in the deployment of common analysis tools and platforms over most major projects. One such tool is Checkmarx, which is deployed across the major projects of the company through a central service new projects can register with. As one of the more complex, slower tools, Checkmarx runs daily, and for each project, yields a list of bugs all of the project developers are responsible for. While rules may vary from project to project, developers are generally instructed to fix recent bugs, and bugs with a higher severity first. The company emphasizes bug fixing right before and after major code releases. Outside of those periods, developers also fix warnings in parallel with their development work.

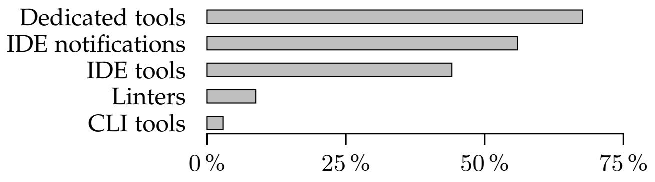


Figure 2: Tool Types Used by Participants (**Q4**).

To answer **RQ1**, we now discuss which tools are used at Software AG, how they are integrated in the development process, and which types of issues they find.

4.1 Analysis Tools

Software AG developers report using a total of 17 different analysis tools that we group by interface types in Figure 2 (**Q4**). *IDE notifications* refer to analyses run by the developers' Integrated Development Environment (IDE) (e.g., uninitialized variables). *IDE tools* designate analysis tools presenting analysis warnings in the IDE (e.g., FindBugs). *Dedicated tools* provide interfaces that are separate from the developer's coding environment (e.g., Fortify, Checkmarx, or CodeSonar). *CLI tools* provide a Command-Line Interface (CLI). We can see that Software AG uses a wide variety of analysis tools. Among the survey participants, 67.6% use dedicated tools, which conforms with Software AG's policy of using such tools in their projects. Participants also receive analysis information from IDE notifications (55.9%) and IDE tools (44.1%). Overall, 36.8% of the participants use only one analysis tool, and 48.5% use only one type of analysis tool.

With **Q5**, we observe that the use of analysis tools is spread over the software development lifecycle: 55.9% of the participants run their analysis tools at coding time, 52.9% during nightly builds, 29.4% at commit time, and 17.6% at major project milestones. We attribute this behaviour to the different types of analysis tools—IDE notifications typically run lightweight analyses and can run at coding time, while longer running tools are not usually able to do so.

We see that Software AG puts efforts in raising awareness about the use of analysis tools and exposes its developers to large system of tools comparable with other large companies studied in past research, thus making Software AG a good case study for evaluating developer behavior and motivation towards static analysis tools.

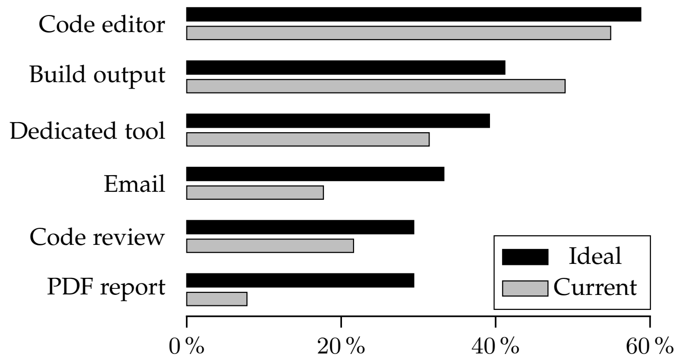


Figure 3: Ideal Reporting Locations that Developers Want Compared to Those of Current Analysis Tools (**Q10**–**Q11**).

4.2 Integration in the Development Process

Once the analysis tools have run, they display warnings in various places, as shown in grey in Figure 3 (**Q10**). Reports in the code editor, build output, and dedicated tools are expected from the tool types that are most used at Software AG. When asked which reporting media they would prefer to use (**Q11**, black bars in Figure 3, participants confirmed wanting to use their current reporting platforms). However, alternative means were requested to a much higher extent: separate PDF files were requested 3.75× more than currently used, email reports were requested 1.89× more, and the code review platform was requested 1.36× more. We attribute this higher demand to the ability of those media to aggregate results from multiple analysis tools in one place, which makes it easier to have an overview of the analysis results. Although we cannot confirm this claim with our current dataset, it is partially supported by **Q12** where 5.5× more developers indicated that they would prefer having the results of multiple analysis tools into one reporting place rather than in different ones (74.5% against 15.7%).

The survey responses show that 82.4% of the participants typically fix analysis warnings themselves (**Q9**). Once the warnings are fixed, they are reviewed by colleagues (79.5%), managers (15.9%), or dedicated teams (9.1%). Out of all fixes, 9.1% go unverified (**Q29**).

According to 47.1% of the participants, analysis tools used at Software AG are configured by a dedicated team. However, 36.8% wrote that they configured some of their analysis tools themselves, and 16.2% that some of their tools run on default settings (**Q6**). We see that a high number of developers set up their own tools themselves, which we attribute to the use of tools not proposed by the global company effort. This behavior, along with the responses to **Q17** where 78.3% of the developers said that they use analysis tools because it "helps me code better" against only 30.4% because of "company policy", suggests that Software AG generally encourages the use of analysis tools and spreads awareness among its developers about the importance of fixing bugs and security vulnerabilities.

4.3 Analysis Warnings

In their responses to **Q7**, participants indicated the warning types that are reported by analysis tools in their projects (grey bars in Figure 4). The warnings that are most reported

Table 1: Conditional Probability of a Warning Type (S = security, P = performance, M = memory, C = concurrency, CS = coding style, FB = functional bugs) Given a Tool Type (**Q7**).

	S	*P*	*M*	*C*	*CS*	*FB*
Dedicated tools	0.31	0.06	0.08	0.04	0.34	0.16
IDE notifications	0.26	0.06	0.04	0.06	0.4	0.19
IDE tools	0.27	0.05	0.06	0.09	0.31	0.21
Linters	0.27	0.09	0	0	0.36	0.27
CLI tools	0.50	0	0	0	0.25	0.25

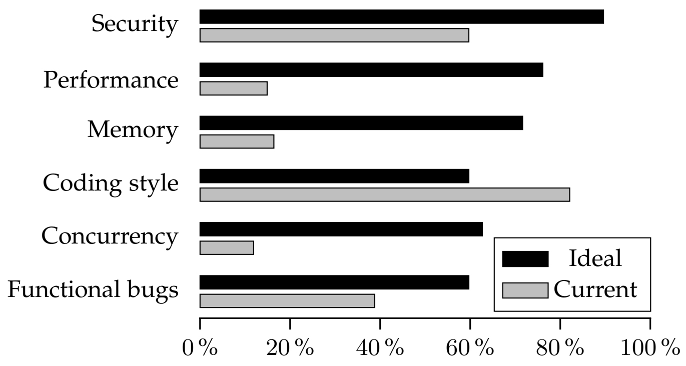


Figure 4: Warning Types that Developers Want Compared to Those Reported by Current Analysis Tools (**Q7**–**Q8**).

are coding style-related (according to 82.1% of the participants), followed by security vulnerabilities (59.7%), and functional bugs (38.8%). Table 1 presents the conditional probabilities of warnings being of a certain type given the tool type. We see that for each tool type, the warning types most likely to be reported are security vulnerabilities, coding style issues, and functional bugs, matching the distribution of warnings that code developers listed as most often reported. Dedicated tools and CLI tools are most likely to report security vulnerabilities, and linters, IDE notifications and IDE tools are more likely to report coding style issues.

When asking developers which types of warnings they would like analysis tools to report (**Q8**, black bars in Figure 4), we observe that the warning types are more distributed. While security vulnerabilities and functional bugs are still high (89.6% and 59.7%, respectively), the number of participants asking for other types of warnings (performance, memory, and concurrency) is higher than the number of participants getting access to such warnings by factors of 4.4× to 5.3×. Finally, participants wish to see less of the most frequently reported warnings: coding style.

Summary (RQ1)

- The most frequently used analysis tools are dedicated tools, as well as IDE tools and notifications.
- Tools report warnings in various locations, but developers prefer to have them aggregated in a central interface.
- Unlike prior studies, Software AG developers rank coding style warnings low on their list, and have a higher interest in performance, memory, and concurrency bugs.

5 Contexts for Using Analysis Tools

Here, we address **RQ2** by detailing when and how Software AG developers use analysis tools, and expand on the reasons that make them use those tools. We focus on

Table 2: Conditional Probability of the Length of a Working Session Given the Tool Type (**Q22**).

	< 10 min	*10 – 30 min*	*> 30 min*	*hours*
Dedicated tools	0.16	0.52	0.13	0.1
IDE notifications	0.31	0.35	0.12	0.12
IDE tools	0.40	0.44	0.04	0.04
Linters	0.75	0	0	0.25
CLI tools	0	1	0	0

Table 3: Conditional Probability of Fixing a Warning in a Certain Time Period Given the Tool Type (**Q13**).

	mins	*< 1 hr*	*< 1 day*	*< 1 wk*	*< 1 month*
Dedicated tools	0.15	0.15	0.38	0.21	0.06
IDE notifications	0.19	0.26	0.35	0.10	0.30
IDE tools	0.23	0.19	0.42	0.12	0
Linters	0.50	0.25	0	0	0
CLI tools	1	0	0	0	0

the tool types that developers use the most: IDE tools and notifications, and dedicated tools.

5.1 Developer Workflow

Although the usage of analysis tools is distributed evenly during the day (morning 11.4%, afternoon 13.6%, and evening 11.4%), the largest group of developers (22.7%) use analysis tools in their spare time, i.e., when they have a few minutes between meetings, or spare hours during the work day (**Q19**). Analysis tools are used frequently in the work week: 75.6% of the participants say they use them multiple times in a week, 24.5% of whom use them more than once a day (**Q18**). This usage pattern indicates that working with analysis tools is not a large task that requires a developer to block a part of their schedule, but instead a set of short tasks that can be interrupted and resumed later. This observation is further supported by responses to **Q13** and **Q22** where we see that the median time for one working session with an analysis tool lasts for 10–30 minutes while the median time for fixing a single warning is between an hour and a day. We can thus infer that in many cases, developers spread their treatment of a warning over multiple working sessions.

The length of a working session with IDE notifications, IDE tools, and dedicated tools mainly vary between a few minutes to 30 minutes (Table 2). While the session length for IDE notifications and tools is evenly distributed between < *10 min* and *10–30 min*, dedicated tools clearly lean towards the longer end of the spectrum. The typical fix times for one warning are shown in Table 3. We find the same trend over the time span of minutes to under a week: with IDE notifications and tools, a warning is fixed in around a shorter time (between an hour and a day) than with dedicated tools (around a day). This trend is explained by the fact that analyses running in the IDE must be able to yield results in a matter of seconds, which restricts them to fast and simple-to-compute checks. Their warnings are thus relatively easier to fix. We see that for the individual tool types, working sessions are typically shorter than the time to fix a warning, the only exception being CLI tools.

Table 4: Goals When Opening Analysis Tools (**Q23**).

	Goal	*% of Devs*
O1	Fix all warnings	36.4%
O2	Fix warnings in a given time	31.8%
O3	Consult warning list	31.8%
O4	Fix a set number of warnings	9.1%
O5	Fix warnings up to a certain standard	4.6%

Table 5: Reasons for Closing Analysis Tools (**Q24**).

	Reason	*% of Devs*
C1	Finished fixing everything	45.5%
C2	Professional obligation	25%
C3	Wait for the analysis tool to update	18.2%
C4	Office distraction	13.6%
C5	Never close the tool	13.6%
C6	Cannot fix an issue	9.1%

5.2 Developer Motivation

Having determined the usage context of analysis tools, we now explore the reasons why developers start and stop using analysis tools in their daily work.

When asked for the reasons why they use analysis tools in general (**Q17**), 30.4% of the participants reveal that it is because of company policies, and 21.7%, that it is because they help them code faster. In addition, 78.3% of the developers report that the tools help them code better, showing that developers value the use of analysis tools outside of company obligations.

Table 4 shows that, independent from the tool type, most of the reasons for which developers open an analysis tool revolve around fixing warnings, with the variation of how many warnings they aim to fix (**Q23**). Conditional probabilities show that a relationship exists between tool types and fixing goals. When using dedicated tools, participants mostly aim at fixing as many warnings as possible in a given time ($Pr = 0.31$), which is a sensible strategy when dealing with complex warnings. With all other tools, participants mainly aim to fix all warnings when they open the tool. Consulting the list of warnings is a frequent reason why developers open the tools for all tool types ($0.24 \leq Pr \leq 0.28$). Table 5 details why developers close an analysis tool (**Q24**), independent from the tool type. The main reason is that they finished fixing all warnings, which we attribute to the use of lighter analysis tools that yield easier-to-fix warnings. The second cause is time limit. The third one is that they wait for the tool to re-run the analysis on the fixed code (complex analyses can take minutes to hours to process an update). A minor reason for which developers close the tool is that they cannot fix a warning, an issue that is likely encountered when dealing with complex warnings that are not properly explained by the tool.

Table 6 shows that, regardless of the reason why the tool was opened, a popular reason for closing it is that all warnings were fixed (**C1**). However, when developers open the tool with a certain limit in mind (time or number of warnings), fixing all warnings is not the main closing reason. Developers are also likely to close the tool due to professional obligations (e.g., a meeting) or waiting for a tool update. We also see that when developers open the tool with the intention to fix all warnings, they only manage

Table 6: Conditional Probability of Reasons for Closing an Analysis Tool given the Reason for Opening it. The Legends for Ox and Cx are Found in Table 4 and Table 5.

	C1	*C2*	*C3*	*C4*	*C6*
O1	0.45	0.05	0.15	0.05	0.15
O2	0.22	0.33	0.17	0.17	0
O3	0.35	0.25	0.15	0.10	0.05
O4	0.20	0.20	0.20	0.20	0

to reach their goal 45% of the time. Otherwise, they are either stuck on a warning or waiting for a tool update. This suggests that when developers do not have time constraints, they have a fair chance to eventually run into warnings that they cannot fix in one working session.

Summary (RQ2)

- Developers mostly use analysis tools in their spare time, and fix warnings in short working sessions.
- Time limitations are the main reason that developers close analysis tools, which imposes different interaction experiences on them when they use the tools (e.g., when choosing warnings to fix).

6 Strategies for Fixing Analysis Warnings

We now answer **RQ3** by exploring developer behavior when fixing warnings, in particular how they prioritize which ones to investigate first, what they do with warnings that they do not understand, and how they collaborate with their colleagues to fix them.

6.1 Prioritizing Warnings

Before they start to fix warnings, developers must first select which warnings to fix. To help them choose, analysis tools typically provide them with additional information such as warning type (e.g., SQL injection), code location, or severity. Table 7 shows the four main strategies adopted by the survey participants when choosing which warnings to investigate (**Q34**). One of the most popular is to prioritize the warnings by impact, which aligns with Software AG's policy of addressing all of the most severe warnings before a major release. This confirms the findings of Vassallo et al. [21], who also mark severity as the most important factor in choosing warnings to fix first. A developer will also preferentially work on warnings that impact their own code or that they know how to fix, because they have the necessary knowledge to do so. The last strategy is to go from the top down in the warning list, which is a sensible methodology for simple lists of warnings that are all fixable within one working session. Such strategy is also useful for longer, more complex lists that the tool already sorts by importance, which is often the case in dedicated tools.

To gain a deeper understanding of which warnings are fixed first, we studied the analysis reports of Checkmarx on two projects: Application 1 and Application 2. The top half of Figure 5 shows the number of warnings found for six of their sub-projects, grouped by confidence (false positive or true positive, as labeled by the developers when they handled the warnings), over the time span of a few months for Application 1 to nearly two years for Application 2. The bottom half shows the same data, grouped by severity, as provided by Checkmarx.

Table 7: Strategies to Prioritize Which Warnings to Address First (**Q34**).

Strategy	*% of Devs*
Prioritize warnings affecting the developer's code	46.3%
Prioritize warnings with the most impact	43.9%
Prioritize warnings the developer can fix	31.7%
Follow the order of the warning list	31.7%

Except for Application 1 G, only a fraction of the warnings are labeled by developers as true or false positives. We see that the variations of the number of labeled warnings follows the variations of the general number of warnings, suggesting that developers actively handle new warnings but usually only look at a fraction of the total number of warnings, or do not often label warnings. We also observe that developers tend to keep the number of warnings with a high severity at a minimum: the plots consistently remain close to 0, confirming our survey results (**Q34**: developers tend to fix warnings with the most impact). This observation is supported by Table 8: the probability of a high severity warning to be in the *to verify* list is very low compared to other types of warnings, and the high severity warnings that remain are most often false positives. Confirmed true positives are handled similarly: they are kept to a minimum and eventually removed, (e.g., Application 2 C).

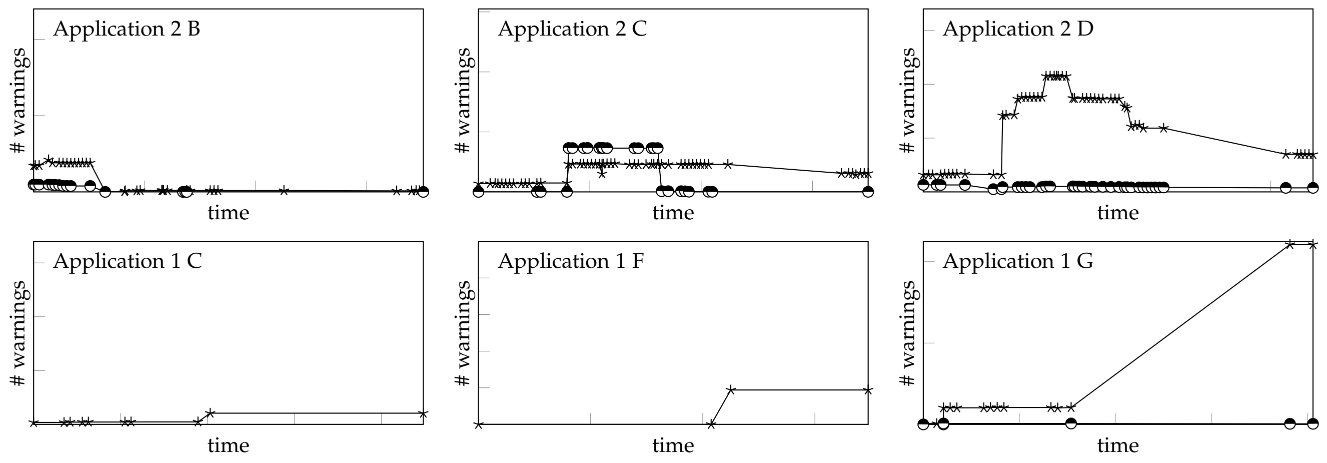
Most projects also have a small number of low severity warnings, which we attribute to the relative ease of fixing such warnings, matching the developer strategy of fixing what they know they can fix. For example, *unchecked return value*, and *null pointer dereference*, likely to be classified with a low severity with a probability of 1, are simple to fix.

In the longer-running Application 2 projects, we also observe that the number of warnings regularly increases and plummets, which (knowing Software AG's release schedule) we suppose with fair confidence corresponds to compliance tests before major product releases or milestones. Outside of those times, the number of warnings decreases slowly due to continuous work done by developers on their spare time, as we have previously discussed.

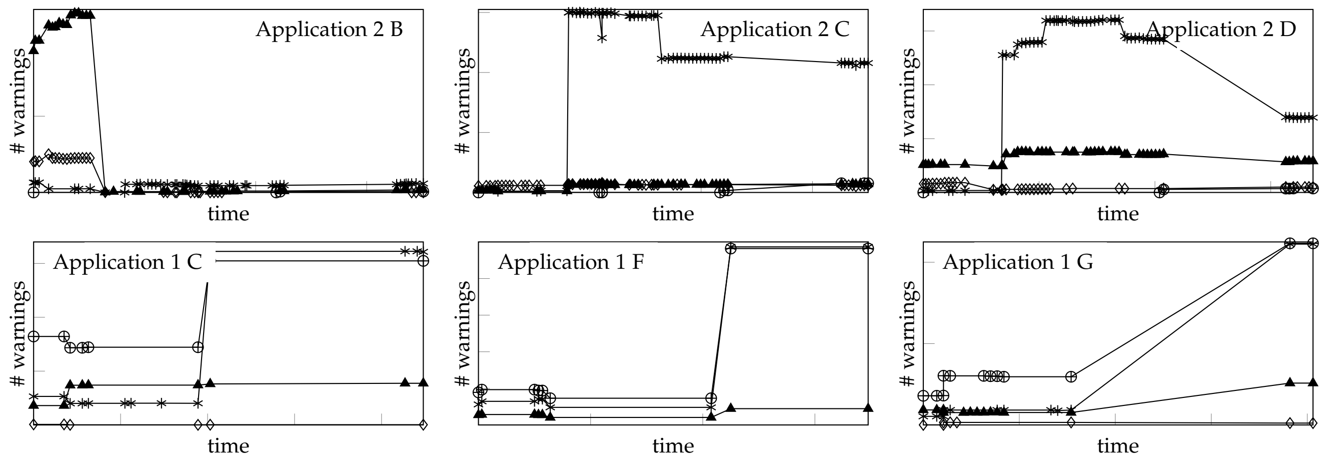
6.2 Detecting False Positives

To help developers decide whether a warning is a false positive or a real issue, analysis tools often provide them with additional metrics. For example, the main screen of Checkmarx allows developers to filter analysis warnings by warning type (e.g., cross-site scripting), code location, severity, and other editable information (e.g., confidence).

Table 9 shows five main strategies that participants use to evaluate if a warning is a false positive (**Q33**). The main strategy is looking at the warning type. This strategy can be a quick way of triaging through warnings, because certain warning types are more often labeled as false positives than others. For example, *memory leak* or *use of uninitialized variable* are very likely to be marked as false positives ($Pr = 0.8$ and $Pr = 0.71$, respectively). However, this strategy can be misleading: without investigating each warning in detail, true positives may be overlooked.



(a) Confidence as Labeled by the Developers (—✶— false positive, —⊖— true positive).



(b) Severity (—◇— high, —▲— medium, —✶— low, —⊕— info).

Figure 5: Number of Warnings for Three Sub-Projects of Application 2, and Three Sub-Projects of Application 1. At the request of Software AG, we do not disclose the axis labels. All axes are in linear scale, the y-axes all start at 0. For the same sub-project, the maximum value on the y-axes are the same on both graphs. The total number of warnings for a Sub-Project can be obtained by summing the plots of its bottom graph, since Checkmarx automatically labels all warnings with a severity.

The second-most popular strategy is related to misinterpretations of some code constructs by the analysis (for example, the behavior of specific libraries or objects). Such constructs seem to be known by developers, and are used to differentiate true from false positives. Another 43.9% of the participants said that they recognize false positives because the warning witness is incorrect, meaning that the analysis' interpretation of the code's runtime behavior is faulty. This situation requires from the developer to investigate the warning in detail, which is a very accurate, but time-consuming strategy. On average, participants investigate 65.1% of the warnings in detail (min = 20%, max = 100%, $\sigma = 26.1$) (**Q32**). The last strategy is to mark as false positives warnings that go through areas of the code that are never executed. While this strategy helps remove false positives, it is insecure to keep vulnerable non-executed code in the codebase, as it could be exploited in the future.

Table 8: Conditional Probability that a Warning is Marked with a Certain Confidence Given its Severity.

	False Positive	*True Positive*	*To Verify*
High	0.92	0.03	0.04
Medium	0.15	0.03	0.83
Low	0.05	0.01	0.93
Info	0	0	1

Table 9: Strategies to Detect False Positives (**Q33**).

Strategy	*% of Devs*
Categories of issues are known false positives	43.9%
Code constructs are not handled by the analysis	39.0%
Warning witness is not executable	31.7%
Code locations are never executed	22.0%
Conditions along the warning are never true	12.2%

Table 10: Why Warnings are Difficult to Understand (**Q36**).

Reason	*% of Devs*
Unfamiliar with the issue	48.8%
Explanation given by the analysis tool is unclear	48.8%
Span over too much of the code base	31.7%
The code base is unclear	4.9%

Table 11: Ranking Developer Actions for Warnings That They Do Not Understand (**Q37**).

Action	*% of Devs*	*Behaviour type*
Leave for later	56.1%	Neutral
Ask for help	51.2%	Good
Ignore	14.6%	Bad
Research and fix	7.3%	Good
Suppress	7.3%	Bad
Escalate	2.4%	Good

6.3 Understanding Warnings

To understand if a warning is a true positive, if they should prioritize it, and how they could fix it, developers have to gain an understanding of the warning. We have previously discussed that developers often use heuristics over easily accessible data to make a decision, because they cannot spend time investigating all warnings. Therefore, the ability of the analysis tool to explain the warning and showcase relevant data is key to supporting its users.

On average, participants understand 36.1% of the warnings they investigate (min = 0%, max = 80%, $\sigma = 32.6$) (**Q35**). To explain this low number, we asked participants for the reasons why warnings can be difficult to understand, which we detail in Table 10 (**Q36**). Three major reasons stand out. First on the list is that the warning is new to the developer, so they need to learn a lot: what the warning means, how it applies to their code, and how to safely fix it in the context of their code. Another reason is that the tool's explanation is unclear. While some tools simply give a generic description of the warning type, others, such as Checkmarx, provide more detailed information, yet it is still difficult to completely explain to developers how the analysis reasons about the warning, especially when the warning is complex and spans over a wide part of the codebase, which leads us to the third reason: the size of the warning. While some analysis warnings can affect small parts of the code (e.g., use of potentially dangerous function), others can involve larger parts (e.g., an SQL injection going from a frontend form to the database).

Table 11 details the treatment of warnings that developers do not understand (**Q37**). Overall, we see three main types of behavior appear: neutral, positive, and negative. We define positive behaviour as actions that bring the developer closer to fixing the warning, negative actions as ones that make the warning harder to be fixed in the future, and neutral as actions that have no incidence on the future treatment of the warning. A majority of the developers (56.1%) adopt the neutral behavior of leaving the warnings for later. More negative solutions are to ignore or suppress the problematic warning. Respectively 14.6% and 7.3% of the developers admit to using them, which should be discouraged. Other participants opt for positive actions and spend more time asking for help, escalating, or researching the warning. On average, participants ask for their colleagues' help for 27.8% of the warnings (min = 0%, max = 70%, $\sigma = 42.2$) (**Q38**).

Table 12: Why Developers Ask for Help (**Q39**).

Reason	*% of Devs*
Others have experience with the code base	46.3%
Others have experience with the type of issue	39%
Others have experience with the analysis tool	31.7%
The developer does not ask for help	14.6%

When developers ask for help (**Q39**), they are interested in the three particular aspects that we discussed in **Q36**: the issue, the codebase, and the analysis tool (in particular, what the tool means when explaining the warning), as seen in Table 12. The first three aspects confirm our observations from **Q36**. In particular, with the second one, we see that of the warnings developers ask about, 46.3% are due to codebase issues, while only 4.9% of the participants find warnings confusing due to lack of understanding of the relevant codebase. We infer that developers rarely ask about confusing warnings, and that a large fraction of the ones they ask about are due to codebase clarity issues. This finding confirms the need for better warning explanations, especially with respect to information about the issue and the analysis tool, which is less at hand to the developers than codebase information they can ask colleagues about. Lastly, 14.6% of the participants do not ask for help. We suppose that this behavior could be caused by time constraints, discouragement of working on a warning for too long, or the social consequences of admitting that they do not understand the warning.

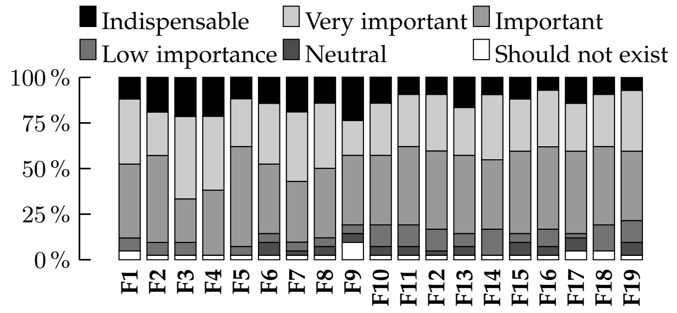
Summary (RQ3)

- Developers tend to choose warnings that they know they can fix, typically through their knowledge of the code base, their experience of the tool, and warning types. Warning recommender systems should thus take into account tool usage context and developer experience.
- When handling a warning, developers use heuristics derived from their experience with the tools. Effective heuristics such as unhandled code constructs should be integrated into the analysis. Weak heuristics such as warning categories can result in negative treatments such as silencing critical warnings.
- The UI of an analysis tool should encourage good behaviour such as collaboration between developers and building a knowledge database.

7 Tool Features

In light of the developers' motivations identified in the previous sections, we discuss which features are of most interest in the user interface of an analysis tool, and how to present them to the developer, answering **RQ4**.

From CLI interfaces to standalone applications to IDE tools, the static analysis tools used at Software AG offer a wide choice in interfaces. To understand the developers' preferences with respect to the UI of analysis tools, we asked them which kinds of layouts they often use (**Q25**). Of all developers, 70.5% use the default layout of the tool, 18.2% use their own custom layout, 4.6% use the company layout, 4.6% change the tool layout according to their needs of the



F1	Responsiveness of the analysis
F2	Responsiveness of the UI
F3	Explain how a bug works and what it is about
F4	Explain how severely the bug can affect the code
F5	Explain how the bug can be executed in the code
F6	Explain how the warning is related to other warnings
F7	Explain how the bug can be fixed, on a high level
F8	Explain how the bug can be fixed in the code
F9	Quick fixes
F10	Visualisation of the individual warnings in the code
F11	Dashboard of the warnings
F12	Search and sort through warnings
F13	Select which bugs to fix first (prioritization)
F14	Keep a list of the developers' bugs
F15	Tracking progress (clear view of what was achieved so far)
F16	Configurability of the analysis
F17	Provide feedback to the tool
F18	Customization of the analysis rules
F19	Collaboration options (with colleagues)

Figure 6: Importance of Analysis Tool Features (**Q30**).

moment, and 2.3% do not use a particular Graphical User Interface (GUI) and stick to CLI tools only. We see that even though Software AG provides developers with a company-specific interface, they prefer the tools' default layouts.

In **Q26**–**Q28**, we asked participants which UI features they often look at when opening, using, and closing an analysis tool. Developer attention is most attracted to the dashboard (a high-level summary of the project's health) when opening the tool (47.7%) and to the warning list when closing it (59.1%). The warning list is central at all times, in particular, when using the tool (68.2%), showing the importance of the information conveyed in this list: the issues, where they are located, and how much still needs to be achieved to meet the company's standards.

For **Q30**, we identified a set of 19 UI features from commercial (e.g., Checkmarx [11], CodeSonar [2], etc.), academic (e.g., FlowDroid [26], Cheetah [27], etc.), and open-source (e.g., FindBugs [15], IntelliJ's Code Inspection [28], etc.) analysis tools, and from past work on their usability [6], [19], [22], [29], and asked participants to rank their importance between six categories: *should not exist*, *neutral*, *low importance*, *important*, *very important*, and *indispensable* (**Q30**). We refer to those features as **F1**–**F19**, all listed in Figure 6. **F1**–**F2** focus on the responsiveness of the tool, **F3**–**F6** address different aspects of explainability, **F7**–**F9** deal with fixing warnings, **F10**–**F12** aim at visualizing the project's health, **F13**–**F15** help keep track of individual warnings, **F16**–**F18** concern analysis configuration and feedback, and **F19** focuses on collaboration.

Figure 6 shows that the most popular features are **F3** (explain a bug), **F4** (bug severity), **F7** (explain how the bug can be fixed), and **F9** (quick fixes). The first three are marked as *very important* or higher by respectively 28, 26, and 24 developers. **F9** has 10 developers marking it as *indispensable*. The popularity of **F3** and **F4** echoes our findings from Section 6: since developers are more interested in severity and understanding the warning, those two features are most important to them. **F7**, which explains how the warning can be fixed on a high level, is highly appreciated. However, **F8**—which does the same but gives more specific recommendations with regards to the codebase—receives less support. Although we cannot be certain, it is possible that manually verifying a fix generated by the analysis would add to what developers currently have to understand, and the risk of introducing more potential bugs in the codebase is too high. Those reasons would also explain the low ratings of **F9**, which has the highest score for *should not exist* with 4 developers compared to an average of approximately 1. **F9** is thus among both the most popular features (supposedly for its gain of time) and the least popular ones.

Features such as collaboration (**F19**), customization of the analysis rules (**F18**), and visualization features (**F10**–**F11**)—which we have identified in Section 6 as ones that can potentially enhance the user-experience of the developers—have received the lowest ratings by a margin of four developers or fewer when compared to the average, despite **F10** and **F11** being often used by the developers (**Q26**–**Q28**). With our current data, it is difficult to say whether those lower ratings are caused by the developers disliking such features in their current analysis tools, them disliking the general idea behind those features, or, if having not experienced those features yet, they are wary of them. On the other hand, all features in the survey were deemed *important* or more by at least 75% of the participants, showing that when designing an analysis tool, even the least popular features would be worth including.

Summary (RQ4)

- Developers most often look at the dashboard when they first open an analysis tool. When using and closing the tool, they most often look at the warning list.
- **F3**, **F4**, **F7**, **F9** are the most popular features among developers.

8 Discussion

In this section, we discuss the outcomes of our study and identify recommendations for building and using analysis tools for software companies such as Software AG.

With **RQ1**, we have seen that Software AG developers are involved with a variety of analysis tools at all stages of the software development process, the most frequently used being dedicated tools, IDE tools, and IDE notifications. Warnings are reported in various places across the working tools, which is consistent with the case study done at Microsoft [19], but developers indicate that they would prefer a common interface for all analysis warnings. Centralizing warnings in the same user interface would reduce developer effort in switching between different interfaces.

Different types of tools are more likely to find different types of warnings. The most reported warnings are about security vulnerabilities, coding style, and functional bugs. In addition, developers would like to obtain information about performance, memory, and concurrency bugs. A major difference from the Microsoft study is the requirements for coding style (separated as "Style" and "Best practices" in [19]). While second most asked in their list, it is significantly less required by Software AG developers, which we attribute to two factors: their tools report too many such warnings, and with time, developers need more help with complex properties of the code than shallower ones.

RQ2 reveals that Software AG developers use analysis tools at short points in time, mostly in their spare time. For them, fixing warnings is a continuous task spread over short working sessions, whose length depends on the tool type. Developers spend less time fixing warnings from IDE notifications and tools, and they have shorter working sessions with those tools. More complex warnings produced by dedicated tools have longer fix times, and cause developers to work with the tools for a longer period of time.

In turn, this makes time limitations the main reason for stopping to use an analysis tool. Time limitations generate different working goals: while developers most often open a tool with the intention to fix warnings, they choose to fix different sets of warnings depending on their available time. This constraint introduces different interaction experiences with the tools, namely how to support developers in choosing which warnings to fix in the given time, which we explore in the following section. As a result, when designing an analysis tool—and modeling the workflow of a user within the tool, we recommend taking into account how long the user intends to use the tool for in a single session.

More minor causes for developers ending a working session are explainability issues for complex warnings, and the long time taken to update analysis results, which have been reported in past studies [4], [6], [19].

With **RQ3**, we study developer behaviour when using the tools, and find that when choosing which warnings to fix first, they aim for those they know they can fix, or for the ones with the most impact. The order suggested by the tools is secondary to that, which we attribute to the time constraints that have previously discussed. As a result, developers base their judgement on their knowledge of the codebase, their experience of the warning types, and warning severity. While current recommender systems—which highlight to the developer warnings they should fix in priority—mainly center around severity, they should also take into account other contextual factors such as the length of a working session, the developer's experience, or—as suggested by Vassallo et al. [21]— the tool in which the analysis is integrated.

To distinguish false positives from true positives, developers often use heuristics derived from their common experience, which can be sound or flawed. Allowing development teams to integrate sound heuristics in the analysis or its configuration would help bridge the gap between how the analysis understands the codebase and how the developer does. Flawed heuristics stem from warnings that are ill-explained by the analysis tool, as confirmed by previous studies [4], [5], [6], which can result in negative warning treatments such as inappropriate silencings or dismissals.

Facilitating explainability is not only restricted to finding better explanations than the ones already provided by the tool. When asking for help from other colleagues, developers seek knowledge about the analysis tool, the warning type, or the codebase. Again, we recommend taking the developer experience into account by, for example, suggesting to a stuck developer the name of a colleague who might have the knowledge they seek, or by building a knowledge database to look up warning into. In addition to using side-channels for asking for help and looking up warnings, we recommend analysis tools to take into account the usage context of a company with a community of developers to encourage collaborative positive behavior.

Focusing on tool features, **RQ4** reports that when first opening an analysis tool, Software AG developers often look at a dashboard to get a general overview of the warnings in the codebase. When closing the tool and while using it, they most often look at the warnings list, which seems to be their primary working tool. We have identified 19 concrete features for static analysis tools, all of which should be considered when designing an analysis tool.

Having identified how software developers use analysis tools in industry and what motivates their behavior, we summarize the outcomes of our study through ten design recommendations for static analysis tools:

1) *Time constraints* are the primary concern of developers when using an analysis tool. The length of a working session with the tool should be taken into account when designing the workflow of an analysis tool. For example, depending on how fast a developer has been in the past on a certain type of warning, the tool can propose suitable warnings to fix for a given span of time.
2) Linked to time constraints, the lack of *responsiveness* of some analysis tools is a user-experience issue encountered by developers at Software AG. The analysis responsiveness and the tool interface should be crafted to minimize waiting times.
3) Static analysis relies on a set of "rules" describing the analyzed code and how to analyze it. As they work with analysis tools, developers build project-specific or company-specific heuristics to deal with warnings faster, some of which can be translated to analysis rules. From the customization of existing rules (as already done by some commercial analyzers) to more intelligent learning of which rules are more useful than others [30], developers should be allowed to *contribute their heuristics to the analysis*.
4) Other heuristics can be harmful, and can be avoided by improving the *explainability* of analysis warnings. Explaining a warning revolves around three knowledge bases: past exposure to certain warning types, knowing the codebase, and knowing the analysis tool.
5) The *developer knowledge* mentioned above should be integrated in *recommender systems* to provide users with personalized warning suggestions, given their abilities and their working time.
6) Developer knowledge should also be made available to all users, through *collaboration options* in the analysis tool, to provide more official alternatives of communication than the currently used side-channels. Examples in the more general field of crowdsourcing range from Q&A platforms like StackOverflow to real-time collaborative code editors [31].

7) More generally, analysis tools should be designed to *encourage good behavior* in situations when users are blocked. Tool and interface features (e.g., Section 7) should be designed with this in mind.
8) In practice, while it is generally recommended to use multiple static analysis tools in conjunction and recoup their warnings together, we also recommend adding *different types of tools* to cover aspects such as performance and memory.
9) When using different tools, we recommend the use of *a single reporting platform* to handle all warnings. As shown by IBM research's platform Khasiana [32], it would uniformize of the usage of analysis tools across the company, help new projects set up their analysis systems, and help developers understand warnings more easily by using information from different sources. Beyond that, we argue that such a platform would be the ideal place for centralizing developer knowledge and fixing warnings collaboratively. The platform would also be able to record developer usage statistics to be reused when planning fixes.
10) The success of analysis tools directly depends on the company's backing. *Policies* enforcing the use of analysis tools and spreading awareness, should be maintained and developed, with for example, initiatives and trainings to sensitize developers towards good behavior when using analysis tools.

9 Threats to Validity

Our study is limited to Software AG and therefore does not necessarily generalize to every software company. However, the participants showed a large diversity in experience and programming languages, and have years of experience working with many static analysis tools at Software AG (**RQ1**–**RQ2**). Thus, the results of our survey reasonably generalize to similar companies using analysis tools.

Some survey questions could be misunderstood. To minimize such errors, we ran a pilot survey with five developers from Software AG. During the data extraction process, we did not find any responses that we could interpret as a response to a misunderstood question.

Another threat to validity is the subjective interpretation of the free-text (**Q15**, **Q31**, and **Q40**) and "Others" survey responses when we reclassified them in different categories. However, for this survey, we did not use the free-text questions, and we verified the classification of the "Others" responses with two raters, with 100% agreement. The survey questions and anonymized responses are available online [25].

While Application 1 and Application 2 do not represent all of Software AG's projects, they are major projects, and are contributed to regularly. We included all of their subprojects in our study, which offers diversity in terms of project type, target platforms, and exposure to Checkmarx.

10 Conclusion

Through a developer survey and an analysis of two projects at Software AG, we have drawn a picture of how static analysis tools are used in industry. Looking at the developer's perspective, we saw that Software AG widely uses analysis tools at all points of the software development process, and that its developers mostly use analysis tools in their spare time to fix warnings. Time constraints heavily influences the developers' goals, workflow, and interactions with the tools, and should be considered when designing an analysis tool.

Developers create internal knowledge and strategies to prioritize warnings, determine if they are true or false positives, and decide how to handle them. Based on this knowledge, we have identified ten recommendations for designing and using analysis tools, such as integrating user-specific knowledge into the warning recommender system, allowing users to encode good heuristics in the analysis, setting up systems to discourage negative behaviour and encourage positive behaviour such as collaboration and customization, areas which are generally overlooked in current static analysis research.

We have more concretely identified 19 desirable features for analysis tools, outlined the importance of company policies to encourage their use, and described the need for a central analysis platform. With this study, we advocate for a more user-centered approach of designing static analysis tools, in which usage context and user motivation can offer a different design perspective and yield new requirements.

Acknowledgments

We thank Software AG for making this work possible. This research has been partially funded by the Heinz Nixdorf Foundation, by the BMBF within the Software Campus initiative, the DFG project RUNSECURE, the NRW Research Training Group on Human Centered Systems Security (nerd.nrw), the Canadian Institute for Advanced Research (CIFAR), and the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] Checkstyle, "Checkstyle home page," https://github.com/checkstyle/checkstyle, Accessed in 2019.
[2] Grammatech, "Codesonar home page," https://www.grammatech.com/products/codesonar, Accessed in 2019.
[3] F. Software, "Fortify home page," http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/, Accessed in 2019.
[4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010. [Online]. Available: http://doi.acm.org/10.1145/1646353.1646374
[5] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead, "Does bug prediction support human developers? findings from a google case study," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 372–381.
[6] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 672–681.
[7] N. Ayewah and W. Pugh, "A report on a survey and study of static analysis users," in *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, ser. DEFECTS '08. New York, NY, USA: ACM, 2008, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1390817.1390819
[8] T. Lowdermilk, *User-Centered Design A Developer's Guide to Building User-Friendly Applications*. O'Reilly Media, 2013.

[9] U. D. of Health & Human Services, "User-centered design basics," https://www.usability.gov/what-and-why/user-centered-design.html, Accessed in 2019.

[10] Software AG, "Software ag," https://www.softwareag.com, Accessed in 2019.

[11] Checkmarx, "Checkmarx home page," https://www.checkmarx.com/, Accessed in 2019.

[12] A. Møller and M. I. Schwartzbach, "Static program analysis," https://cs.au.dk/ amoeller/spa/spa.pdf, Accessed in 2019.

[13] Synopsys, "Coverity home page," https://scan.coverity.com/, Accessed in 2019.

[14] Research Triangle Institute, "The economic impacts of inadequate infrastructure for software testing," 2002.

[15] H. Shen, J. Fang, and J. Zhao, "Efindbugs: Effective error ranking for findbugs," in *International Conference on Software Testing, Verification and Validation (ICST)*, 2011, pp. 299–308.

[16] Open Web Application Security Project (OWASP), "Source code analysis tools," https://www.owasp.org/index.php/Source_Code_Analysis_Tools, Accessed in 2019.

[17] CERN Computer Security, "Static code analysis tools," https://security.web.cern.ch/security/recommendations/en/code_tools.shtml, Accessed in 2019.

[18] K. Vorobyov and P. Krishnan, "Comparing model checking and static program analysis: A case study in error detection approaches," *SSV*, 2010.

[19] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 332–343. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970347

[20] L. Layman, L. Williams, and R. S. Amant, "Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, Sep. 2007, pp. 176–185.

[21] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall, "Context is king: The developer perspective on the usage of static analysis tools," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 38–49.

[22] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, Sep. 2008.

[23] 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 *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, May 2017, pp. 334–344.

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

[25] L. Nguyen Quang Do, J. R. Wright, and K. Ali, "Artifacts for "why do software developers use static analysis tools?"," https://sideresearch.wordpress.com/, Accessed in 2019.

[26] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 259–269. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594299

[29] L. Nguyen Quang Do and E. Bodden, "Gamifying static analysis," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*

[27] N. Q. D. Lisa, A. Karim, L. Benjamin, B. Eric, S. Justin, and M.-H. Emerson, "Just-in-time static analysis," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 307–317. [Online]. Available: http://doi.acm.org/10.1145/3092703.3092705

[28] JetBrains, "Intellij home page," https://www.jetbrains.com/idea/, Accessed in 2019.
2018, Lake Buena Vista, FL, USA, November 04-09, 2018. New York, NY, USA: ACM, 2018, pp. 714–718. [Online]. Available: https://doi.org/10.1145/3236024.3264830

[30] L. Nguyen Quang Do and E. Bodden, "Explaining static analysis with rule graphs," *IEEE Transactions on Software Engineering*, 2020.

[31] M. Yuen, I. King, and K. Leung, "A survey of crowdsourcing systems," in *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*, 2011, pp. 766–773.

[32] 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 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 99–108. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810310

Lisa Nguyen Quang Do is a doctoral researcher in the 'Secure Software Engineering' group at Paderborn University. She has received her M.Sc. in Computer Science from EPFL in 2014. Her research focuses on improving the usability of analysis tools for code developers and analysis developers through different aspects that range from the optimization of the analysis algorithm to the implementation of its framework to the usability of its interface.

James R. Wright is an Assistant Professor in the Department of Computing Science at the University of Alberta and a Canada CIFAR AI Chair through the Alberta Machine Intelligence Institute. He received his PhD from the University of British Columbia in 2016. His overarching scientific agenda is to model multi-agent learning and behavior, with a particular focus on human behavior.

Karim Ali received his PhD degree from the University of Waterloo in 2014. He is currently an Assistant Professor in the Department of Computing Science at the University of Alberta. His research interests are in programming languages and software engineering, particularly in scalability, precision, and usability of program analysis tools. His work ranges from developing new theories for scalable and precise program analyses to applications of program analysis in security and Just-in-Time compilers.