Noname manuscript No.
(will be inserted by the editor)

Using Code Reviews to Automatically Configure Static Analysis Tools

Fiorella Zampetti · Saghan Mudbhari · Venera Arnaoudova · Massimiliano Di Penta · Sebastiano Panichella · Giuliano Antoniol

Received: date / Accepted: date

Abstract Developers often use Static Code Analysis Tools (SCAT) to automatically detect different kinds of quality flaws in their source code. Since many warnings raised by SCATs may be irrelevant for a project/organization, it can be possible to leverage information from the project development history, to automatically configure which warnings a SCAT should raise, and which not. In this paper, we propose an automated approach (Auto-SCAT) to leverage (statement-level) code review comments for recommending SCAT warnings, or warning categories, to be enabled. To this aim, we trace code review comments onto SCAT warnings by leveraging their descriptions and messages, as well as review comments made in other different projects. We apply Auto-SCAT to study how CheckStyle, a well-known SCAT, can be configured in the context of six Java open source projects, all using Gerrit for handling code reviews. Our results show that, Auto-SCAT is able to classify

Fiorella Zampetti
University of Sannio
E-mail: fiorella.zampetti@unisannio.it

Saghan Mudbhari
Washington State University
E-mail: saghan.mudbhari@wsu.edu

Venera Arnaoudova
Washington State University
E-mail: Venera.Arnaoudova@wsu.edu

Massimiliano Di Penta
University of Sannio
E-mail: dipenta@unisannio.it

Sebastiano Panichella
Zurich University of Applied Sciences
E-mail: panc@zhaw.ch

Giuliano Antoniol
Ecole Polytechnique de Montreal
E-mail: antoniol@ieee.org

code review comments into CheckStyle checks with a precision of 61% and a recall of 52%. While considering also the code review comments not related to CheckStyle warnings Auto-SCAT has a precision and a recall of $\approx$ 75%. Furthermore, Auto-SCAT can configuring CheckStyle with a precision of 72.7% at checks level and a precision of 96.3% at category level. Finally, our findings highlight that Auto-SCAT outperforms state-of-art baselines based on default CheckStyle configurations, or leveraging the history of previously-removed warnings.

Keywords static analysis tools · code reviews · automated tool configuration

1 Introduction

Static Code Analysis Tools (SCATs) are, for software developers, a precious complement to more expensive quality assurance techniques, including testing and code inspection (Beller et al., 2016; Vassallo et al., 2018; Zheng et al., 2006). For instance, SCATs enable the early identification of potential bugs, security vulnerabilities, performance issues, or deviations from the project/organization coding guidelines. The latter can be particularly relevant to enforce consistency. Different developers contributing to a project may have different programming styles, such as different ways of naming identifiers, using indentation, using braces, *etc.* Failing to follow appropriate coding guidelines may result in a hard to read and to maintain source code (Bacchelli and Bird, 2013; Beller et al., 2014) or the rejection of patches of newcomers contributing to the project.

However, SCATs feature a large variety and number of checks. For instance, CheckStyle[1] features over 150 checks. Many of such checks could be irrelevant for a given project or development context (Panichella et al., 2015; Vassallo et al., 2018), *e.g.,* some coding guidelines may not be in use in a given organization, meaning that developers handle different warning categories depending on the specific development context and rely on specific factors when selecting warnings to fix (*e.g.,* team policies and composition). As a result, SCAT could produce a large number of irrelevant warnings (Couto et al., 2013). This leaves developers with a negative opinion and a general skepticism about the usefulness of SCAT (Johnson et al., 2013), which emphasizes the necessity to improve existing strategies for the selection of relevant alarms that are shown to developers.

Properly configuring a SCAT, *i.e.,* enabling relevant checks is not trivial. This is especially true when (i) a project has not a shared knowledge of what flaws should be avoided and what coding guidelines should be followed, (ii) a novice developer is joining a project/wants to contribute to it, and cannot get any insight about the code quality checks to be performed, and (iii) there is need to configure a "centralized" usage of SCATs, *e.g.,* in a Continuous Integration (CI) pipeline (Duvall et al., 2007; Zampetti et al., 2017).

[1] http://checkstyle.sourceforge.net

In this paper, we propose an approach, tailored to CheckStyle, called **Auto-SCAT** (**Au**tomated Configura**to**r of **S**tatic **C**ode **A**nalysis **T**ools) which automatically configures SCATs by leveraging information coming from past code reviews, thus identifying code review comments pointing to potential violations to coding standards that a SCAT could detect. We conjecture that, during a code review task, developers discuss and highlight various quality issues (Fagan, 1976) relevant to the project. Differently from previous works inspired by the program history (Williams and Hollingsworth, 2005), previous warning fixes (Kim and Ernst, 2007), or different kinds of product (code) and process (changes) metrics (Ruthruff et al., 2008), we suppose that review comments provide valuable insight and can be used to automatically configure SCAT. In particular, as reported by Bacchelli and Bird (2013), 30% of code review comments are about source code improvements. Moreover, recent work observed that developers exploit SCATs during code review, *e.g.,* Panichella et al. (2015) found that a significant proportion of warnings highlighted by SCATs are fixed during code reviews. However, over 80% of the warnings are generated by only 20% of the checks (Marcilio et al., 2019). For this reason, if one enables a warning-generating check that is not relevant for a given project or organization, several false alarms will be triggered. This highlights the importance of configuring SCATs in a way that only relevant checks are included.

Auto-SCAT mines and exploits comments made by developers during code reviews on specific source code statements (in the following referred to as "inline comments"), and leverage these comments to enable static analysis checks. For instance, if previous code reviews mention that Javadoc comments are required for methods, or source code lines are too long, then Auto-SCAT automatically enables static analysis checks aimed at detecting these types of problems. In other words, Auto-SCAT is able to configure a SCAT if previous code reviews mention related issues that are considered as relevant for the project. To facilitate the mapping of code reviews comments onto checks, Auto-SCAT leverages a knowledge base consisting of SCAT checks along with their descriptions, and a set of comments coming from other projects and mapped onto such checks. Given the diversity of code review comments, this knowledge base facilitates Auto-SCAT to trace code review comments onto checks.

To evaluate Auto-SCAT, we automatically configure CheckStyle, a widely used SCAT for Java (particularly suited to enforce coding guidelines), for six Java open source projects by leveraging their code reviews performed using the Gerrit[2] code reviewing environment. More specifically, we investigate (i) to what extent the code review inline comments on which CheckStyle raises a warning, are related to coding guidelines violations; (ii) the performance of Auto-SCAT to provide an accurate and complete configuration of CheckStyle's checks; and (iii) how Auto-SCAT compares with different baselines, including default CheckStyle configurations, and configurations based on the CheckStyle execution on the previous project's history.

[2] https://www.gerritcodereview.com

Our results show that Auto-SCAT infers appropriate checks with an overall precision of 75.1% and an overall recall of 74.6%. Moreover, when looking at the accuracy in configuring CheckStyle, the precision of Auto-SCAT is equal to 72.7% when enabling specific checks, while reaches 96.3% when enabling categories. Finally, Auto-SCAT outperforms baselines using default CheckStyle configuration and historical data.

The contributions of this work can be summarized as follows:

– We devise Auto-SCAT, a novel approach that leverages code review comments to configure SCATs.
– We provide an oracle of manually-mapped code review comments onto CheckStyle warnings that the comment pertains to, useful for replications' purposes (Zampetti et al., 2020).

The paper is structured as follows. Section 2 discusses the related literature concerning code reviews and Static Code Analysis Tools. Section 3 describes how Auto-SCAT works, while Section 4 reports the empirical study we conducted. Results are presented and discussed in Section 5, whereas the study threats are detailed in Section 6. Finally, Section 7 concludes the paper and outlines directions for future work.

2 Related work

This section discusses relevant literature concerning (i) code reviews, (ii) static code analysis tools, and (iii) SCAT warnings prioritization.

2.1 Code Reviews

Previous literature has investigated Modern Code Review (MCR) practices. Some of them examine which factors affect the code review response time and outcome (Baysal et al., 2013; Bosu, 2014; Weißgerber et al., 2008). For instance, Bosu (2014) found that changes submitted by core members have higher chances of being accepted very fast. Other studies have highlighted that MCR can be used for sharing and transferring knowledge mainly for educating newcomers or building a strong community (Bacchelli and Bird, 2013; Beller et al., 2014; Bosu et al., 2017; Rigby et al., 2008). From a different perspective, Kononenko et al. (2016) analyzed the code review quality perception of developers, highlighting that the review quality is mainly impacted by the feedback provided during the code review process. Finally, some works investigated the impact of MCR on the overall software quality in terms of the likelihood of introducing bugs (Bavota and Russo, 2015; McIntosh et al., 2016) and anti-patterns (Morales et al., 2015). Finally, recent work by Pascarella et al. (2018) empirically studied the information needs in MCR such as knowing the uses of methods and variables declared/modified in the code under review.

Differently from the studies reported above, our work leverages the code review inline comments linked to SCAT checks to verify whether they contain enough information for automatically configure SCATs in a way that only relevant/actionable warnings are displayed to developers.

From a different perspective, researchers have also looked at how code review was used to identify defects in source code. In particular, Mäntylä and Lassenius (2009) studied what kinds of defects are found in code reviews of 9 industrial and 23 student projects. They found that the majority of defects (71%) are evolvability problems, while only 21% of defects found in code reviews are functional ones. A follow-up study was conducted by Beller et al. (2014), to look at the kind of code review problems being fixed. Their study confirmed results of Mäntylä *et al.*, with a 75:25 ratio between evolvability and functional problems. Our work focuses on evolvability problems, *e.g.,* those Mäntylä *et al.* call textual representation (*i.e.,* naming and comments), visual representation (*e.g.,* indentations and bracket usage), and distribution of organization (*e.g.,* long, complex, or dead code) defects.

Summarizing, the aforementioned corpus of research contributed to achieve a better understanding of what are the goals achieved by developers during code reviews. Auto-SCAT does not explicitly cope with issues related to potential bugs, and not even to the presence of code smells/needs for refactoring. Instead, by recommending the activation of CheckStyle issues, our work contributes to help code reviewers to achieve a more consistent adherence to coding styles.

2.2 Static Code Analysis Tools

Previous literature has investigated the extent to which and how developers use SCATs during software development (Vassallo et al., 2018). Specifically, Johnson et al. (2013) analyzed the reasons why developers do (not) use SCATs, identifying that false alarms, the large volumes of warnings and the inadequate understandability of the output generated by SCATs are responsible for their under-usage. Spacco et al. (2006), instead, identified a relationship between the warnings removal and their priority (*i.e.,* developers do not remove warnings with low priority). While looking at the relation between fault occurrence and SCAT warnings, Couto et al. (2013) found that there is no static relationship. The latter is partially contradicted by Zheng et al. (2006) who found that SCATs play a significant role in identifying security vulnerabilities. Finally, Querel and Rigby (2018) combined static code analysis and statistical bug models to identify when risky commits introduce warnings. Their approach, *i.e.,* WARNINGSGURU, reduces the overall number of commits and warnings developers have to examine to identify bugs.

A recent work by Beller et al. (2016) analyzed the usage of SCATs in open source, showing that their adoption is not widespread, but also that only in $\simeq 5\%$ of the cases their configurations do not correspond to the ones provided by default. Zampetti et al. (2017), instead, studied the usage of SCATs in Continuous Integration, highlighting that build breakages (i) are mainly related to adherence to coding standards, and (ii) are quickly fixed by actually solving the problem, rather than by disabling the warning. Finally, Panichella et al. (2015) studied the usage of SCATs in MCR processes, finding that during code review the removal of some warnings is very high, *i.e.,* from 50% up to 100%. Their results point out the possibility to use the source code change history for configuring SCATs.

Summarizing, previous work has shown how developers use different types of SCATs, including tools for code style checks, *e.g.,* CheckStyle. Moreover, as reported by Zampetti et al. (2017), such tools are even used to make a build fail. Therefore, a proper configuration, reflecting the organization/project coding standards is desirable, instead of simply activating all possible checks from the default tool's configuration.

2.3 Approaches for SCATs Warnings Prioritization

Previous literature has identified approaches aimed at prioritizing and classifying warnings generated by SCATs to deal with the challenge of too many false alarms and non-actionable warnings. Williams and Hollingsworth (2005) implemented an approach that searches for a commonly fixed bug and refines its results through historical information. Kim and Ernst (2007), instead, defined a history-based warning prioritization algorithm by mining warning fix experience recorded in the software change history, improving warning precision up to 67%. Finally, Ruthruff et al. (2008) identified 33 factors (most of them relying on the change history of the project) that may relate to actionable warnings, and used logistic regression and screening methodology to identify actionable warnings. Their approach predicts false positive warnings over 85% of the time and actionable warnings over 70% of the time. However, a shortcoming of using historical information is that some warnings can disappear as a consequence of source code being deleted or moved to a different location. For this reason, the usage of historical information can lead to imprecision if used for configuring SCATs.

Differently, our approach looks at inline comments posted by developers during code reviews highlighting possible violations to coding standards and guidelines, without accounting for whether developers change the code to address the warnings. Hence, our approach does not look at removed warnings, while it enables checks for warnings felt as relevant by developers.

Machine learning algorithms have been used to minimize displayed non-actionable warnings (Hanam et al., 2014; Reiss, 2007; Yoon et al., 2014; Yüksel and Sözer, 2013). For instance, Yoon et al. (2014) used Support Vector Machine (SVM) with $\simeq$ 87% of accuracy, while Hanam et al. (2014) used alert characteristics along with an a priori knowledge about which code patterns are actionable, for ranking alerts according to the likelihood that they are actionable. Moreover, a recent study by Ribeiro et al. (2019), investigated the possibility of combining the reports generated by different SCATs to identify the issues least likely to be false positives. Specifically, they constructed a classifier to rank static analyzer alarms based on the probability of a given alarm being an actual bug in the code.

Complementary to prioritization algorithms, there are also approaches that cluster similar/related warnings (Fry and Weimer, 2013; D. Zhang and Zhang, 2013; Muske et al., 2013) to enable developers skipping all warnings in a cluster if they think that it does not contain relevant warnings. Finally, there are works aimed at modifying the user interface for assisting developers in finding actionable warnings among the ones raised by SCATs (Anderson et al., 2003; Ayewah and Pugh, 2009; Khoo et al., 2008; P. Cousot et al., 2005; Phang et al., 2009).

The aforementioned approaches help developers in identifying what are the warnings to consider. Our perspective is different since we want to learn SCAT configurations, avoiding to generate irrelevant output for developers even before the SCAT is being used/adopted. A more important difference is in the kind of tool the different approaches are able to prioritize. Existing approaches prioritize bug finding tools based on their capability to find bugs that have been previously fixed. However, this approach cannot work for style checking tools such as CheckStyle. This is because, during software development, code style issues are addressed without explicitly opening issues. Instead, those are problems typically highlighted and addressed during code reviews. This is why our work leverages code review comments to decide upon the activation of CheckStyle checks.

3 Auto-SCAT Usage Scenario and Approach Description

In the following, we start describing the scenario in which Auto-SCAT can support developers, and then we describe the Auto-SCAT approach.

3.1 Auto-SCAT Usage Scenario

The typical usage scenario of Auto-SCAT reflects a scenario described in the Duvall et al. (2007) book on continuous integration (Chapter 3, page 58, scenario "Coding Standard Adherence"). The scenario describes what happens when a new developer joins a project, but s/he is not suitably following coding guidelines. To this extent, providing developers with written guidelines (assuming these are available) does not work because developers may not bother to read them. Therefore, Duvall *et al.* suggest integrating static analysis tools in the continuous integration pipeline to generate a build failure every time source code not meeting coding style guidelines is submitted. Such build failures warn developers on the need to fix those violations as soon as possible, and represent valuable support for the code reviewing activity (Cassee et al., 2020).

In this context, how can projects leverage Auto-SCAT? Let us consider that no formalized guidelines for code style exist in a project, while developers

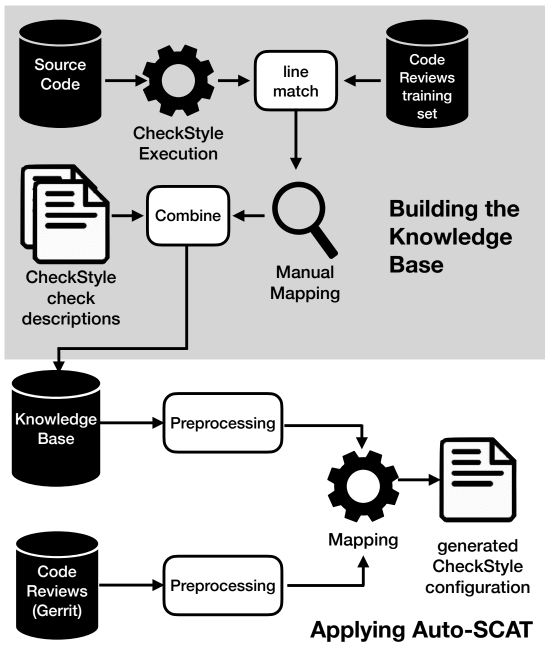


Fig. 1 Overview of Auto-SCAT.

have previously performed code reviews, pointing out code style issues. Auto-SCAT learns from code reviews, and automatically configures static analysis tools (CheckStyle in the current implementation) enabling checks relevant for the project.

3.2 Auto-SCAT Approach Overview

In the following, we describe how Auto-SCAT automatically generates configurations for CheckStyle. However, it is possible to apply the same approach to other SCATs and to comments from other code review infrastructures.

Fig. 1 provides an overview of the approach. As mentioned in the introduction, Auto-SCAT leverages (i) a knowledge base, made up of CheckStyle checks, along with their descriptions, and code review comments (from several projects different from the one to configure) manually mapped onto checks; and (ii) past code reviews of the project on which we apply Auto-SCAT.

To create a CheckStyle configuration for an unseen project, Auto-SCAT uses code reviews comments of the new project by computing the textual similarity of each new comment with comments stored in the knowledge base; if the similarity is above an activation threshold, the corresponding check is flagged as relevant and activated in the CheckStyle configuration. Note that, if the similarity threshold is the same for multiple checks, they all get enabled. Since CheckStyle groups checks into categories dealing with specific issues, *e.g.,* naming of identifiers, Javadoc comments, or indentation, it is possible to have similar code review comments belonging to different checks in the same category. For this reason, Auto-SCAT generates configurations at two levels of granularity, at check-level (*i.e.,* each check is enabled or disabled separately) and category-level (*i.e.,* the configuration enables or disables the whole category).

The accuracy and completeness of the Auto-SCAT recommendations depend on (i) the code review comments in the knowledge base, and (ii) the availability of code review comments for the project to configure. The former aids to map diverse code review comments related to specific issues, *e.g.,* different reviewers may provide comments with different wording to report a problem such as the need for improving identifier naming or splitting a long line. The latter, instead, is a necessary precondition for the applicability of Auto-SCAT. Indeed, a check is enabled only if a related issue has been previously mentioned in the projects' code reviews.

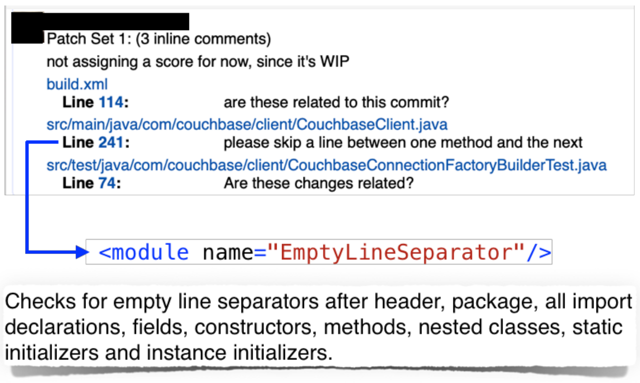
Depending on the role and the development context of the SCAT, one may opt for favoring precision over recall. Obviously, for SCATs identifying likely defects, it might be useful to favor recall over precision, although previous research has warned about the limited adoption of SCATs due to low precision (Johnson et al., 2013; Wedyan et al., 2009). At the same time, for tools enforcing coding style guidelines, such as CheckStyle, a high number of irrelevant warnings may discourage developers in fixing them. Therefore, maximizing recall while sacrificing too much precision may not be the best option. Thus, in our work, we decided to address the problem of having a low precision assuming that, in the case of CheckStyle alarms, developers are highly interested in having a high precision in the recommendations.

In the following subsections we provide details on how (i) comments are preprocessed, (ii) the knowledge base is created, and (iii) the activation threshold is selected.

3.2.1 Code review preprocessing and representation

To create an automated classification for the SCAT (CheckStyle in our case), we start by preprocessing code review comments extracted from Gerrit, using a typical Information Retrieval (IR) normalization process (Baeza-Yates and Ribeiro-Neto, 1999).

We preprocess comments by removing special characters, splitting compound words (identifiers) using the camel case heuristic, and applying stop-words removal and stemming. We have explored three stop-words configurations: retaining stop-words, removing stop-words, and retaining stop-words only if they appear in the name of a CheckStyle warning (*e.g.,* we retain "after" since it appears in the "WhiteSpaceAfter" check). The latter is the one used by Auto-SCAT. As for stemming, Auto-SCAT uses the Porter stemmer (Porter, 1980). As last step, comments are mapped into a Vector Space Model (VSM) (Salton et al., 1975) using the *tf-idf* weighting scheme (Baeza-


[redacted] Patch Set 1: (3 inline comments)

not assigning a score for now, since it's WIP

build.xml

Line 114: are these related to this commit?

src/main/java/com/couchbase/client/CouchbaseClient.java

Line 241: please skip a line between one method and the next

src/test/java/com/couchbase/client/CouchbaseConnectionFactoryBuilderTest.java

Line 74: Are these changes related?

```
<module name="EmptyLineSeparator"/>
```

Checks for empty line separators after header, package, all import declarations, fields, constructors, methods, nested classes, static initializers and instance initializers.

Fig. 2 Example of inline code review mapped onto a CheckStyle check.

Yates and Ribeiro-Neto, 1999) by using the Gensim (Řehůřek and Sojka, 2010) library.

3.2.2 Auto-SCAT Knowledge base definition

Auto-SCAT requires a knowledge base, *i.e.,* a set of code review comments manually traced onto CheckStyle checks.

Fig. 2 shows a code review inline comment[3] from the *Couchbase* project, which is mapped onto the `EmptyLineSeparator` check. In the bottom, we report the check description as extracted from the CheckStyle user manual[4].

In principle, one may wonder *why we require a knowledge base, and why we do not just textually compare the code review comments of the project to be configured with CheckStyle check description.* More in detail, the code review comments should be (textually) related to such descriptions, and this would have permitted an unsupervised configuration approach. We explored such a possibility on the same dataset used in our empirical evaluation obtaining a very low precision ($\simeq$ 26%). This suggests that the way code quality issues are pointed out in code reviews is different from the way CheckStyle describes checks about the same code quality issues. For such a reason, we move towards the idea of building a check description corpus with code reviews of other projects, as detailed in the following.

To create the knowledge base, we first download a set of code reviews, using the Gerrit's API. After that, we store the inline code review comments and the line number in the source file where the code review comment is. As it would not be feasible to manually classify a large set of inline comments, we use CheckStyle to select only those lines that 1) have one (or more) review associated comment(s); and 2) cause CheckStyle to raise a warning. Since the goal of running CheckStyle is to identify all warnings that may relate to a

[3] http://review.couchbase.org/#/c/21805/ line 241, author's hidden for privacy reasons.
[4] http://checkstyle.sourceforge.net/checks.html

code review comment, we configure CheckStyle so that it does not miss any warning. Specifically, we include all checks in our configuration except those requiring project-specific configuration parameter (*e.g.,* "Regexp" — a check to detect strings in source code matching a regular expression). For checks requiring a parameter (*e.g.,* the maximum number of parameters allowed for each method declaration belonging to the *ParameterNumber* check), we set low thresholds so that we do not miss potentially relevant warnings.

CheckStyle warnings raised in one line may not be related to a code review comment made on the same line. Thus, we created a sample of inline comments posted on source code lines where CheckStyle raises a warning and manually validate them, flagging as relevant and adding to the knowledge base only those comments discussing quality issues identifiable by CheckStyle. However, during our manual validation, we have encountered many comments that were simply reporting misbehavior that had nothing to do with the checks identifiable by CheckStyle. As an example, the comment: *"I'm not sure what semantic impact this would have. The modification simply checks before casting which should be always be done prior casting. You[r] suggestion would mean to not perform any code in such a case which not necessarily needs to be correct"* in EclipsePlatformUI highlights the presence of behavior that deviates from the expected one. Other cases, instead, were too much general like *"Accepted second proposal"* in EclipsePlatformUI or *"I'm not entirely convinced but made the change. Probably something that should be discussed further"* in Vaadin. Finally, we also found comments that were pointing out the presence of code smells that cannot be detected by CheckStyle. For instance, in Vaadin we found a comment stating: *"seems redundant"*, and a different comment reporting: *"should perhaps extract and rename the interface from ShortcutActionHandler as no longer specific to it"*. All the inline code review comments above have been assigned to a category (*i.e.,* "Not related to CheckStyle") mainly aimed at grouping comments which body is not related to any CheckStyle checks/categories.

Just relying on a set of manually-classified comments is not always a viable solution due to the varying distribution of the number of code review comments mapped to CheckStyle checks, which can be in some cases very low. To cope with this problem, we create three configurations of the knowledge base, and investigate which one leads to the best performance:

1. **Comments only:** the knowledge base only contains code review comments that highlight quality issues detectable by CheckStyle;
2. **Comments + descriptions:** the same as the previous point augmented with the check description from the CheckStyle documentation. If no code review comments are belonging to a check, then the check is not included in the knowledge base;
3. **Comments + descriptions, or just descriptions:** similar to (2), however if there are no comments for a check, then the check is simply represented by its description.

3.2.3 Check activation threshold

Auto-SCAT uses textual similarity and an activation threshold to determine whether to activate a check for a new project. The rationale is that, if a reviewer has made a comment similar to a check (or similar to a comment traced to a check in the knowledge base) then the check needs to be enabled. To this aim, we applied the preprocessing step and VSM, thus computing the cosine similarity (Huang, 2008) between each review comment of the new project and the textual corpus that represents the check. Specifically, a VSM represents documents as vectors in a multi-dimensional space in which each column is a document and each raw is the relevance of a word in each document. In this representation, words are then providing orthogonal information, *i.e.,* each word is associated to a specific dimension.

Auto-SCAT calibrates the thresholds on the projects on which the knowledge base has been built, *i.e.,* a set of projects different from the one where Auto-SCAT configures CheckStyle (as detailed in Section 4.2).

4 Study Design

The *goal* of the study is to investigate the degree to which code review comments can be used to automatically configure SCATs. The *quality focus* is the Auto-SCAT accuracy in generating a SCAT configuration, as compared to baselines leveraging default SCAT configurations or the SCAT execution over the past project's history. The *perspective* is of researchers and developers, who are interested in reducing the number of irrelevant SCAT warnings to deal with. The *context* consists of code review comments from six open-source Java projects, all using Gerrit as a code review tool, and CheckStyle as a SCAT to be configured. Specifically, the study aims at addressing the following research questions:

RQ_1: *To what extent code review comments occurring on lines where CheckStyle fails are related to quality issues that can be detected by that tool?*
Why: First of all, we need to analyze the extent to which code review comments contain information relevant to our purpose, *i.e.,* whether the reviews discuss issues that CheckStyle detects.
How: We analyze code review comments where CheckStyle raises warnings. Specifically, we manually verify whether the comments pertain to the warnings raised by CheckStyle. This lets us evaluating the percentage of code review comments that can be used to configure CheckStyle.

RQ_2: *How accurate is Auto-SCAT?*
This research question aims at assessing the performance of Auto-SCAT, and is divided into two sub research questions, *i.e.,* $RQ_{2.1}$**:** *How accurate is Auto-SCAT to classify code review comments into CheckStyle checks?*, and $RQ_{2.2}$**:** *How accurate is Auto-SCAT to configure CheckStyle?*
Why: Once we know that code review comments can be used to configure CheckStyle, we want to evaluate how accurately Auto-SCAT maps inline

comments to checks ($RQ_{2.1}$) and how accurately it configures CheckStyle, by enabling or disabling checks ($RQ_{2.2}$).

How: To address $RQ_{2.1}$, we evaluate the Auto-SCAT capability to correctly classify code review comments at both check and category level. We assume that, if Auto-SCAT can correctly link a code review comment to a check (or a category), then we can use this information to enable the check (or the whole category). The rationale of computing the accuracy at category level is that if for a project some relevant checks belonging to a category are specified (*e.g.,* naming conventions), it makes sense to enable the whole category even if there is no specific comment (yet) on the individual checks of that category. To address $RQ_{2.2}$, instead, we evaluate the Auto-SCAT capability to correctly enable or disable CheckStyle checks. In this case we put ourselves in the perspective of a developers that wants to use Auto-SCAT to configure CheckStyle.

While looking at the performance of Auto-SCAT, it is important to highlight that we aim at increasing the usefulness of static code analysis tools (SCATs) by reducing the number of irrelevant warnings reported to developers. Our objective is guided by previous work indicating that, while SCATs have the potential to identify the presence of likely defects or vulnerabilities, they also produce a high number of false positives (Wedyan et al., 2009). However, the great number of irrelevant warnings that SCATs produce may hinder their adoption in the software development process (Beller et al., 2016; Johnson et al., 2013; Zampetti et al., 2017), as well as in all development contexts such as continuous integration, code review, and local programming (Vassallo et al., 2018).

RQ_3: *How accurate is Auto-SCAT compared to a baseline?*

Why: It is important to verify whether Auto-SCAT outperforms a baseline approach.

How: Using as a ground-truth projects' coding standard guidelines, we compare the Auto-SCAT configuration with that of a default configuration and with a configuration based on historical information. As regards the former, our choice is motivated by a previous work by Beller et al. (2016) highlighting how open source projects tend to rely on the default configuration of SCATs rarely adding customized checks. Moreover, they also found that the configurations once added, are only rarely modified mainly within the first week of their appearance. For the historical-based configuration, instead, we execute CheckStyle (by enabling all checks) on previous versions of the project, and determine which warnings have been removed at least once through changes committed by developers.

4.1 Object systems

Table 1 summarizes relevant information for the six projects considered in our study. For each project we report (i) the time interval within which we extracted the code review comments, (ii) the total number of inline code review

Table 1 Studied Dataset (TC =Number of inline code review comments, CW Pairs = Number of comments on lines where CheckStyle raises a warning, SC = Sampled comments for manual validation, TP = Number of comments in the sample that are linked to a CheckStyle check).

Project	Time Frame	TC	CW Pairs	SC	TP
Couchbase	Jan,2012-Dec,2016	753	397	19	10
Eclipse CDT	Apr,2012-Nov,2015	4,855	1,730	423	167
Eclipse JDTCore	Dec,2014-Nov,2016	119	55	5	5
Eclipse Platform UI	Aug,2014-Mar,2017	3,407	838	360	107
OpenDaylight Controller	Jul,2013-May,2017	7,975	2,374	573	292
Vaadin	Nov,2012-Aug,2016	14,485	2,595	492	169
Total		**31,594**	**7,989**	**1,872**	**750 (40.0%)**

comments, (iii) the number of comment-warnings pairs, *i.e.,* the number of inline code review comments left on source code lines where CheckStyle raises a warning, (iv) the total number of pairs used for our manual validation, and finally (v) among the manually-validated pairs, the number of cases where the comment body describes code quality issues identifiable by CheckStyle. All the studied projects have been used in previous work analyzing the use of SCATs in code reviews (Panichella et al., 2015). The dataset described in Table 1 has been used to address the three research questions defined above. However, RQ_3 is addressed using, as a test set, data from a different project, *i.e.,* Mylyn[5]. For the definition of the ground truth, we manually analyze Mylyn's coding guidelines[6] to determine the categories of checks that are relevant to the project.

4.2 Evaluation Methodology and Metrics

To verify whether Checkstyle warnings correspond to issues discussed in code review comments ($\mathbf{RQ}_1$), we investigate the proportion of comments related to issues that are detected by CheckStyle. To this end, we run CheckStyle on the source files where code review comments exist and we manually validate a significant stratified random sample (fifth column in Table 1) of the whole set of code review comments made at a line where a CheckStyle warning is raised. All the comments that are not related to CheckStyle warnings have been labeled as "not related". The manual validation has been performed by two independent annotators, and in case of disagreement, a third person helped to solve the conflict.

To address $\mathbf{RQ}_{2.1}$, we evaluate the Auto-SCAT capability in correctly and completely associating code review comments of a project onto CheckStyle checks, by performing a cross-validation analysis across the six selected projects: we built the knowledge base on five projects and computed the performance metrics on the one left-out.

[5] https://www.eclipse.org/mylyn/

[6] https://wiki.eclipse.org/Development_Conventions_and_Guidelines

Before doing this, we needed to calibrate the similarity thresholds on the projects on which the knowledge base has been built. More specifically, given N projects:

– First of all, one project is removed from the corpus. The knowledge base will contain data coming from N-1 projects, while the remaining (withheld) project plays the role of the unseen project. However, the code review comments from the withheld project play the role of ground truth.
– After that, Auto-SCAT computes the all-by-all cosine similarity between the review comments of the withheld project and the CheckStyle relevant comments in the reduced knowledge base. It is important to remark that for the given projects (i) relevant comments in the corpus are mapped onto checks – ground truth and (ii) it is possible to also access the code review comments not related to CheckStyle.
– Subsequently, Auto-SCAT varies the activation threshold between zero and one at step of 0.1 and, for each threshold, computes the precision and recall. Specifically, the precision is computed cumulatively without distinguishing among single checks. As a result, it is possible to generate a withheld project precision curve as a function of the threshold.
– Finally, we select the threshold that achieves a good compromise between precision and recall.

The type of classification we perform is a single-label multi-class classification. When performing the classification at check-level, a class is a check, while at category-level, a class is a category. We measure Auto-SCAT performance using precision and recall (Sokolova and Guy, 2009), computed over the classifications produced by our classifier for every single class. After that, the precision and recall of the multi-class classifier are computed by averaging the respective metrics for each class to get a better overview of the results. Specifically, we calculate micro- and macro-averaged metrics (Yang, 1999) by using the following equations:

$$M_{micro} = M(\sum_{i=1}^{n} TP_i, \sum_{i=1}^{n} FP_i, \sum_{i=1}^{n} FN_i, \sum_{i=1}^{n} TN_i)$$

$$M_{macro} = \frac{1}{n}\sum_{i=1}^{n} M(TP_i, FP_i, FN_i, TN_i)$$

where n is the number of classes and M is the evaluation metric. Following those formulas, precision, for example, is calculated as follows:

$$P_{micro} = \frac{\sum_{i=1}^{n} TP_i}{\sum_{i=1}^{n}(TP_i + FP_i)}$$

$$P_{macro} = \frac{1}{n}\sum_{i=1}^{n} \frac{TP_i}{TP_i + FP_i}$$

Simply put, macro-averaged metrics are the arithmetic means over the per-class scores. Thus, macro-averaging the results gives equal weight to each class.

To calculate micro-averaged metrics, we simply look at all instances and ignore the class they belong to, *i.e.,* we create a global contingency table. Thus, micro-averaging the results gives equal weight to each classified instance. Both measures are important as they provide different insights. In particular, in the presence of imbalanced classes, micro-averaging the results will highlight the effectiveness of a classifier on the large classes. This will give us an idea of how our approach will perform in a real situation as in reality, the prevalence of different classes is imbalanced. However, macro-averaged results allow us to get a better sense of the effectiveness of the approach on small classes (Manning et al., 2008).

When micro-averaging the performance metrics of a multi-class classifier, the precision and recall are always equal. This is due to the way the average of the performance metrics is computed. Let us suppose that an instance of C_i is misclassified as C_j, *i.e.,* a code review comment that pertains to check C_i is classified into check C_j. This misclassification is a false positive (FP) for C_j and a false negative (FN) for C_i. In other words, each FN concerning C_i is a FP for a class in $C - C_i$. As a consequence, $\sum_{i=1}^{n} FP_i = \sum_{i=1}^{n} FN_i$, meaning that the micro-averaged precision and recall are equal. Moreover, we evaluate the performance, *i.e.,* precision and recall, of Auto-SCAT by comparing the checks (categories) that it enables against the checks (categories) that we expect it to enable from the manually validated code review comments.

Finally, it is important to remark that in this context, we evaluate Auto-SCAT considering three different types of the knowledge base, *i.e.,* (i) Comments only, (ii) Comments + description, and (iii) Comments + description, or just description, finding that the second provides better performance in terms of precision and recall. So the results we provide are related to the usage of Auto-SCAT having a knowledge base that for each check having at least a code review comment, in the projects from which Auto-SCAT is able to learn, contains the code review comments and the description of the check as reported in the CheckStyle documentation.

To address $\mathbf{RQ}_{2.2}$, instead of evaluating the extent to which a code review comment is correctly classified, we evaluate the extent to which a CheckStyle check is correctly enabled or disabled. Therefore, if there is at least one comment activating a check, we consider that check as enabled. As evaluation metrics, we use the same metrics used for $\text{RQ}_{2.1}$.

To address $\mathbf{RQ}_3$, we evaluate the Auto-SCAT capability to configure a project (*i.e.,* Mylyn) which data is “unseen” in the cross-validation reported in RQ_2. As explained in Section 4.1, we manually defined the ground-truth for the Mylyn CheckStyle configuration by inspecting the Mylyn documentation and coding guidelines. The goal of our manual validation is to determine a set of checks/categories that need to be enabled in order to follow Mylyn's guidelines. This allowed us to produce a CheckStyle configuration that meets the coding guidelines of the subject project. Subsequently, we use the above configuration to determine how good Auto-SCAT is in determining a CheckStyle configuration in two different scenarios, *i.e.,* (i) as done in RQ_2, *i.e.,* by leveraging the content of code review comments belonging to the project, or (ii) by directly computing the similarity between those guidelines and the CheckStyle check description, *i.e.,* using guidelines as they were code review comments. The rationale of the latter is to show how Auto-SCAT could be used by leveraging a different source than code review comments, *i.e.,* coding standard guidelines.

More specifically, we compare Auto-SCAT with: (i) two popular default configurations, namely the Sun[7] and Google[8] default CheckStyle configurations and (ii) a historical approach that generates a configuration based on warnings resolved during the code review process. For the historical approach, we compare a patch on which a code review comment is made with the final patch. More specifically, we run CheckStyle on the project commit history, and determine which warnings have been removed over changes at least once. We chose to compare patches with comments as opposed to only the initial patch to take into account any intermediate patches for which quality issues are resolved. For each baseline, we report precision and recall. It is important to point out that the historical approach is different from comparing the Auto-SCAT results with a ground-truth. This is because, with respect to the past, unseen violations types in the source code (and new discussions about checks in the code review) could appear.

5 Results

This section reports and discusses the results answering our three research questions.

5.1 RQ_1: *To what extent code review comments occurring on lines where CheckStyle fails are related to quality issues that can be detected by that tool?*

As shown in the last column of Table 1, about 40% of the code review comments made on source code lines where CheckStyle raises a warning in our sample pertain to code quality issues that can be detected by CheckStyle. A recent work by Panichella and Zaugg (2020) have proposed a more recent taxonomy of code review changes highlighting how the availability of new emerging development technologies (*e.g.,* cloud-based technologies) and practices (*e.g.,* Continuous Delivery) has pushed developers to perform additional activities during modern code review (MCR) and that additional types of feedback are expected by reviewers. Also in the context of this work, code review comments dealing with code improvement and understanding aspects are relevant and frequently addressed by developers, which confirm the relevance of CheckStyle checks.

Moreover, while looking deeper into code review comments not related to CheckStyle checks/categories we found many cases highlighting the need for

[7] `https://checkstyle.sourceforge.io/sun_style.html`
[8] `https://checkstyle.sourceforge.io/google_style.html`

refactoring actions, *e.g., "This refactoring makes sense but doesn't belong as part of this change which is purely about adding generics. We can clean that up in another change."* in Eclipse Platform UI, as well as cases describing a bug that has to be fixed, like *"All these things could be a problem if we launch two launches exactly at the same time. Maybe using a LaunchGroup we could trigger a bug. I'll open a bug about this."* in Eclipse CDT. Our manual investigation also identified cases where the comments are used for better understanding the behaviour of a piece of code, *e.g., "What is the reason that closeNow() doesn't set justClosed and schedule the timer to reset this?"* in Vaadin or cases aimed at verifying the external impact like *"What's the performance impact of this hack? Vaadin"* in Vaadin.

Moreover, out of 154 CheckStyle checks (available in the current version), we found a mapping for 70 of them. Such checks (detailed in Table 2) belong to 12 categories, out of the 14 available categories. The only categories for which we did not find a mapping were *Metrics* (*i.e.,* in the investigated projects developers did not discuss of high cyclomatic complexity, coupling, etc.), and *Regexp*, *i.e.,* customized checks created through regular expressions. The checks that are discussed the most in the studied projects are *JavadocMethod* and *JavadocType*, *i.e.,* those related to the presence of a proper Javadoc documentation for methods and classes/interfaces. Noticeably, *JavadocMethod* is discussed in all the studied projects. Other warnings mostly related to formatting (*e.g., NoLineWrap* and *NeedBraces*) but also to information hiding (*i.e., VisibilityModifier*) are the next most frequently discussed ones. Finally, the less discussed quality issues are mainly related to naming improvement (*e.g., AbbreviationAsWordInName*, *LocalVariableName*, and *MethodName*) and decreasing complexity (*e.g., NestedIfDepth*, *DesignForExtension*, *SimplifyBooleanExpression*, and *MagicNumber*).

RQ$_1$ summary: 40% of the code review comments left on source code lines where CheckStyle raises a warning in our sample, have a body describing and/or pointing out a code quality issues identifiable by code styling tools (*i.e.,* CheckStyle), and are related to a broad variety (45%) of CheckStyle checks, *i.e.,* 12 out of 14 categories. This suggests that, in principle, code reviews comments could be exploited to automatically configure SCAT (*i.e.,* CheckStyle).

Table 2: Auto-SCAT Precision and Recall in code review comments classification

CheckStyle check/category	**TP**	**FP**	**FN**	**Pr(%)**	**Rc(%)**
Annotations	**0**	**0**	**3**	**0.0**	**0.0**
AnnotationLocation	0	0	1	0.0	0.0
SuppressWarnings	0	0	2	0.0	0.0
Block Checks	**17**	**19**	**32**	**47.2**	**34.7**
AvoidNestedBlocks	0	1	2	0.0	0.0
EmptyBlock	0	6	2	0.0	0.0

Continued on next page

Table 2 – *Continued from previous page*

CheckStyle check/category	**TP**	**FP**	**FN**	**Pr(%)**	**Rc(%)**
EmptyCatchBlock	0	1	1	0.0	0.0
LeftCurly	5	8	2	38.5	71.4
NeedBraces	12	1	23	92.3	34.3
RightCurly	0	2	2	0.0	0.0
Class Design	**13**	**20**	**13**	**39.4**	**50.0**
DesignForExtension	0	1	4	0.0	0.0
FinalClass	12	15	4	44.4	75.0
HideUtilityClassConstructor	0	3	5	0.0	0.0
InnerTypeLast	1	1	0	50.0	100.0
Coding	**31**	**20**	**46**	**60.8**	**40.3**
AvoidInlineConditionals	0	4	1	0.0	0.0
DeclarationOrder	0	0	2	0.0	0.0
ExplicitInitialization	4	3	4	57.1	14.3
FinalLocalVariable	1	2	6	33.3	62.5
IllegalCatch	10	3	6	76.9	25.0
MagicNumber	4	1	12	80.0	25.0
MissingCtor	1	1	3	50.0	25.0
MultipleStringLiterals	3	1	4	75.0	42.9
NestedIfDepth	1	0	1	100.0	50.0
OneStatementPerLine	0	1	1	0.0	0.0
SimplifyBooleanExpression	1	2	0	33.3	100.0
UnnecessaryParentheses	5	2	0	71.4	100.0
Headers	**21**	**3**	**5**	**87.5**	**80.8**
Header	21	3	5	87.5	80.8
Imports	**5**	**19**	**13**	**20.8**	**27.8**
AvoidStarImport	0	2	5	0.0	0.0
AvoidStaticImport	1	7	0	12.5	100.0
ImportOrder	0	2	5	0.0	0.0
RedundantImport	0	3	1	0.0	0.0
UnusedImports	4	5	2	44.4	66.7
Javadoc Comments	**91**	**39**	**100**	**70.0**	**47.6**
JavadocMethod	67	23	44	74.4	60.4
JavadocParagraph	0	1	1	0.0	0.0
JavadocStyle	2	2	4	50.0	33.3
JavadocTagContinuation	0	4	2	0.0	0.0
JavadocType	22	4	38	84.6	36.7
JavadocVariable	0	2	10	0.0	0.0
NonEmptyAtclauseDescription	0	3	1	0.0	0.0
Miscellaneous	**49**	**11**	**15**	**81.7**	**76.6**
CommentsIndentation	0	0	1	0.0	0.0
FinalParameters	1	1	2	50.0	33.3
Indentation	19	3	3	86.4	86.4

Continued on next page

Table 2 – *Continued from previous page*

CheckStyle check/category	**TP**	**FP**	**FN**	**Pr(%)**	**Rc(%)**
TodoComment	29	1	2	96.7	93.5
TrailingComment	0	6	7	0.0	0.0
Modifiers	**39**	**11**	**19**	**78.0**	**67.2**
ModifierOrder	2	0	7	100.0	22.2
RedundantModifier	4	3	4	57.1	50.0
VisibilityModifier	33	8	8	80.5	80.5
Naming Conventions	**12**	**32**	**55**	**27.3**	**17.9**
AbbreviationAsWordInName	4	0	2	100.0	66.7
AbstractClassName	1	5	0	16.7	100.0
ConstantName	4	5	2	44.4	66.7
LacalVariableName	0	7	17	0.0	0.0
MemberName	0	1	7	0.0	0.0
MethodName	2	5	18	28.6	10.0
ParameterName	0	4	4	0.0	0.0
TypeName	1	5	5	16.7	16.7
Size Violations	**14**	**1**	**9**	**56.0**	**60.9**
AnonInnerLength	0	1	1	0.0	0.0
LineLength	14	9	7	60.9	66.7
MethodLength	0	1	1	0.0	0.0
Whitespace	**95**	**76**	**53**	**55.6**	**64.2**
EmptyLineSeparator	7	13	1	35.0	87.5
FileTabCharacter	5	1	2	83.3	71.4
GenericWhitespace	7	17	1	29.2	87.5
MethodParamPad	0	1	1	0.0	0.0
NoLineWrap	38	4	4	90.5	90.5
NoWhitespaceAfter	0	9	3	0.0	0.0
NoWhitespaceBefore	1	3	2	25.0	33.3
ParenPad	1	3	10	25.0	9.1
SeparatorWrap	0	2	2	0.0	0.0
SingleSpaceSeparator	1	5	2	16.7	33.3
TypecastParenPad	0	0	1	0.0	0.0
WhitespaceAfter	14	3	14	82.4	50.0
WhitespaceAround	21	15	10	58.3	67.7
Not Related To CheckStyle	**1,010**	**213**	**112**	**82.6**	**90.0**
TOTAL	**1,397**	**464**	**475**	**75.1**	**74.6**

5.2 RQ_2: *How accurate is Auto-SCAT?*

In the following, we report results aimed at showing the Auto-SCAT classification performance, *i.e.,* to what extent code review comments can actually be mapped onto checks ($RQ_{2.1}$) and to what extent CheckStyle checks are

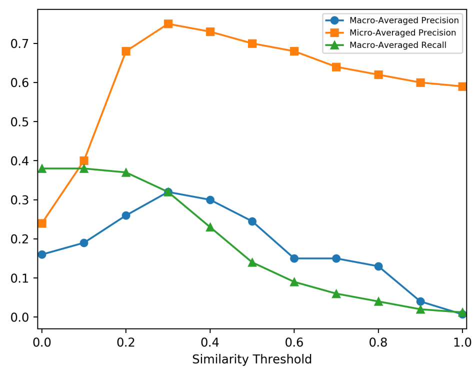


Fig. 3 Performance of Auto-SCAT at check level.

correctly enabled or disabled ($RQ_{2.2}$). As explained in Section 4, given the six projects of Table 1, we build the training corpus on five project, and then we compute the accuracy on the remaining one.

5.2.1 $RQ_{2.1}$: How accurate is Auto-SCAT to classify code review comments into CheckStyle checks?

Fig. 3 reports the micro- and macro-averaged precision and recall of our classifier at different similarity thresholds. Note that, as already highlighted, the micro-averaged precision and recall assume the same value.
As expected, the micro-averaged precision increases as the threshold increases from 0, and after reaching inflection at thresholds 0.3 (74.7% precision) and 0.4 (73.0% precision), it starts decreasing. Specifically, at threshold 0, all the comments are classified into one of the checks leading to low precision (*i.e.,* most of the comments are not related to CheckStyle checks). As we start increasing the minimum similarity threshold, more comments are filtered out as not related to CheckStyle. This results in an inflection point, after which the threshold is so high that many correct classifications will be discarded as not related to CheckStyle since their similarity is lower than the threshold.

Regarding macro-averaged precision and recall, we observe that the recall is (obviously) highest with a threshold of 0, with a low value of precision (16.1%). Such a threshold means that all comments are attributed to a check, which produces a low precision as most comments are either unrelated to CheckStyle. On the opposite side, at threshold 1, both recall and precision are the lowest — 1.3% and 0%, respectively. This is because the comments are not classified into checks unless they are exactly the same to a group of comments of the same class.

As with micro-averaged precision, when we increase the threshold starting at 0, the macro-averaged precision starts increasing reaching a deflection point at 0.3 (31.9%) and 0.4 (29.6%), after which it decreases. The macro-averaged recall has the highest values between thresholds 0 and 0.2 ($\simeq 38\%$) after which it starts dropping quickly until threshold 0.6, reaching a recall of 9.6%.

By looking at Fig. 3, we can conclude that, overall, the micro-averaged precision and recall are higher than the macro-averaged precision and recall. In other words, since large classes dominate small classes in micro-averaging (Manning et al., 2008) and micro-averaged precision and recall are a measure of effectiveness on large classes, unsurprisingly Auto-SCAT is able to classify in a better way code review comments belonging to the majority classes (*i.e.,* checks that have a great number of code review comments in the knowledge base).

To summarize, the most suitable threshold that achieves both high precision and recall is of 0.3, which will be used in the following. Before going deeper on the performance of Auto-SCAT in identifying code review comments mapped onto CheckStyle checks, we will briefly report some examples in order to better understand the reasons why at a threshold of 0.3 and 0.4 Auto-SCAT shows the best compromise between precision and recall. In Eclipse CDT there is a code review comment: *"Avoid cryptic abbreviations"*[9] that with a similarity of 0.39 is properly mapped to the Abbreviation As Word In Name check, while in Vaadin the comment: *"missing javadocs though it is quite obvious what it does"*[10] with a similarity of 0.3 is correctly assigned to the Javadoc Method check.

Table 2 reports the Auto-SCAT precision and recall in identifying code review comments mapped onto different CheckStyle checks and categories by using, as explained before, a similarity threshold of 0.3. More specifically, for each check/category we report the number of true positive, *i.e.,* the number of code review comments correctly mapped onto the check/category, false positive, *i.e.,* the number of code review comments wrongly assigned to a check/category, and false negative, *i.e.,* the number of code review comments belonging to a specific check/category wrongly assigned to a different check/category. Finally, by using the above information, we have computed for each check/category the precision and recall.

By looking at Table 2 we observe that some checks are classified with 0% precision while some are classified with 100% precision. The majority of the checks classified with 0% precision have 5 or fewer instances of code review comments in the textual corpus, except for *JavadocVariable, TrailingComment, LocalVariableName,* and *MemberName.* All the checks with 0.0% precision have a recall of 0.0%. In order to improve the identification of the above checks we conjecture that: increasing the number of projects, and the number of code review comments belonging to them in our textual corpus may help in improving the performance of Auto-SCAT in properly identifying them.

Moreover, results reported in Table 2 highlight how some categories such as Block Checks, Class Design, and Imports are more difficult to be identified

[9] core/org.eclipse.cdt.ui/src/org/eclipse/cdt/internal/ui/refactoring/pullup/PullUpInformation.java+refs-changes-77-22177-3

[10] client/src/com/vaadin/client/widgets/Escalator.java+refs-changes-28-7628-1

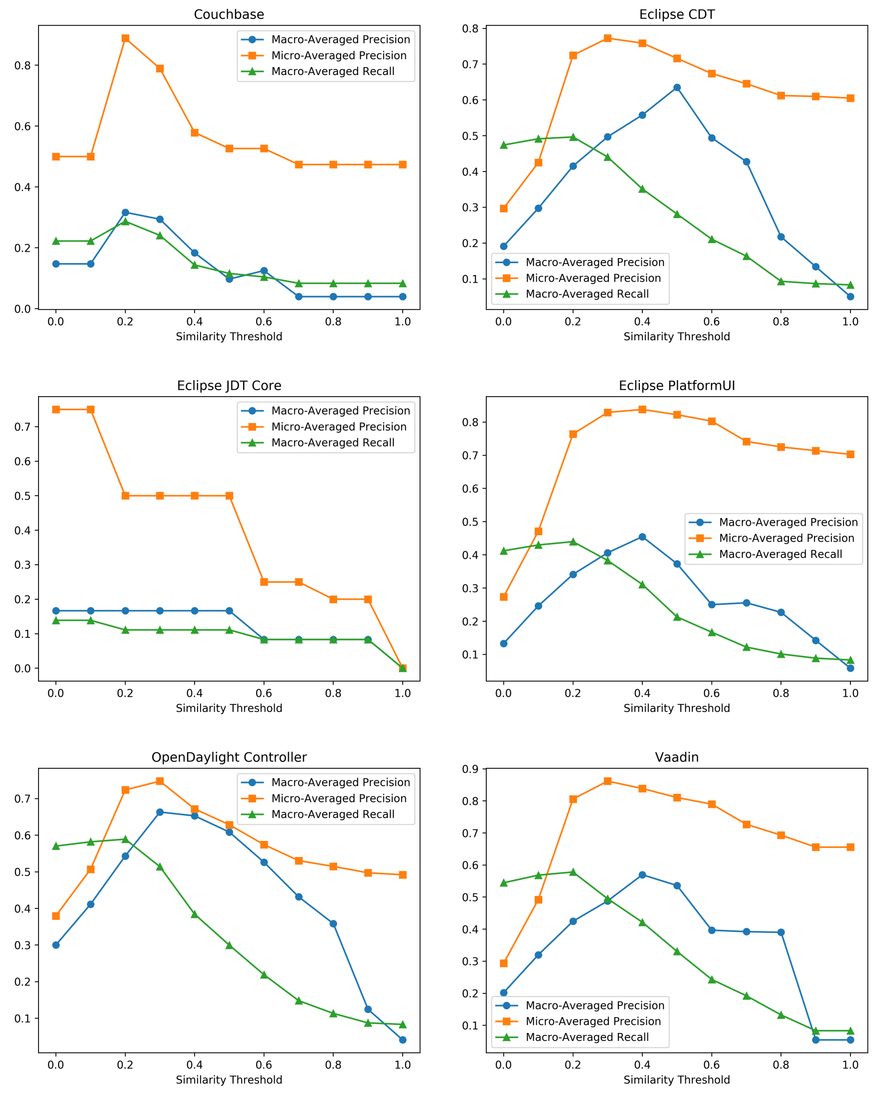


Fig. 4 Per project performance of Auto-SCAT at category-level for different thresholds.

by Auto-SCAT compared to other checks such as Headers and Miscellaneous. The above result is mainly related to the vocabulary used for reporting different type of problems belonging to different types of checks. For instance, the checks in the Miscellaneous category are about substantially different topics meaning that they do not have a common vocabulary. The latter justifies the reason why Auto-SCAT classifies them with high performance. The same does not apply to the Javadoc Comments category where the single checks have a high overlapping in terms of vocabulary. Consider as an example the comment *"javadoc?"* in Vaadin belonging to the Javadoc Method check and the comment *"Please add javadoc like other MICommand"* in Eclipse CDT belonging to the Javadoc Type check. In conclusion, it is possible to state that for Auto-SCAT it is easier to properly classify code review comments that belong to CheckStyle checks that have a unique vocabulary.

Looking at the predominant category, namely Not Related to CheckStyle, we see that Auto-SCAT is able to discriminate those comments from the ones actually mapped onto CheckStyle checks. Indeed, Auto-SCAT for these type of comments provides a precision of 82.6% with a recall equal to 90%. Moreover, if we sum up the performance of Auto-SCAT in identifying code review comments belonging to CheckStyle checks, by excluding data related to the Not Related to CheckStyle category, we can conclude that the performance are still good, considering also the high imbalanced ratio between the above two categories, with an overall precision of 60.7% and an overall recall of 51.6%.

Looking at the overall performance (see the TOTAL row in Table 2) we can state that Auto-SCAT classifies correctly 1,397 over 1,872 code review comments (precision of 75.1% and recall of 74.6%). The latter is mainly due to the high precision obtained while classifying comments not related to CheckStyle.

Fig. 4 shows, for each project, the performance of Auto-SCAT at category-level while varying the similarity threshold. We observe that thresholds 0.3 and 0.4 are optimal for all but two projects — Couchbase and Eclipse JDT Core. This is probably due to the lower number of code review comments related to CheckStyle checks, *i.e.,* 10 and 5, respectively.

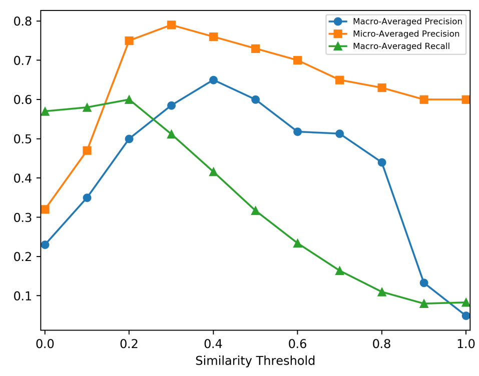


Fig. 5 Overall performance of Auto-SCAT at category-level for all thresholds.

The overall performance of Auto-SCAT when configuring CheckStyle, varying the similarity threshold, at category-level is reported in Fig. 5. Similarly to the results obtained when using Auto-SCAT for classifying the code review comments into specific CheckStyle checks, also here the optimal performance is obtained while using a similarity threshold of 0.3 or 0.4. Specifically, using

Table 3 Performance of Auto-SCAT per project for threshold 0.3 and individual optimal threshold at category-level.

	Threshold 0.3			**Optimal Threshold**			
Project name	**Micro Prec.**	**Macro Prec.**	**Macro Rec.**	**Threshold**	**Micro Prec.**	**Macro Prec.**	**Macro Rec.**
Couchbase	78.9%	29.4%	24.1%	20%	88.8%	31.7%	28.7%
Eclipse CDT	77.3%	49.7%	44.1%	50%	71.6%	63.5%	28.1%
Eclipse JDT Core	50.0%	16.7%	11.1%	10%	75.0%	16.7%	13.9%
Eclipse Platform UI	82.9%	40.6%	38.3%	40%	83.8%	45.5%	31.1%
OpenDaylight Controller	74.7%	66.3%	51.4%	30%	74.7%	66.3%	51.4%
Vaadin	86.2%	48.8%	49.5%	40%	83.9%	56.9%	42.1%
OVERALL (Mean)	**75.0%**	**41.9%**	**36.41%**	—	**79.6%**	**46.8%**	**32.6%**

a similarity threshold of 0.3 we obtain a micro-averaged precision of 79.84% with a macro-averaged precision of 58.51%. Increasing the threshold at 0.4 Auto-SCAT shows a decrease of the micro-averaged precision (76.55%) with an increase of the macro-averaged precision (65.2%).

While comparing the performance of Auto-SCAT at category-level with the one at check-level, we observe a slight increase in the micro-averaged precision, *i.e.,* $\simeq$ 5% with a threshold of 0.3 and $\simeq$ 3.5% for a threshold of 0.4. Moreover, we observe a significant increase in the macro-averaged precision of $\simeq$ 27% and $\simeq$ 36% for thresholds equal to 0.3 and 0.4, respectively.

Table 3 summarizes the performance of Auto-SCAT at category level, when all checks are calibrated at threshold 0.3 for all projects, and when a per-project threshold is chosen. We observe that, except for Couchbase and Eclipse JDT Core, using a threshold of 0.3 leads to similar performance in terms of the micro-averaged precision compared to the case in which the thresholds are tuned per-project. As shown in Table 1, these are also the projects with the smaller number of code reviews (and consequently, of manually validated code reviews), therefore it is possible that, in these cases, such low value may depend to chance. Looking at the macro-average precision, instead, the performance obtained while using the thresholds tuned per-project is slightly better than the one obtained using a fixed threshold of 0.3.

Table 4 Macro-averaged Precision and Recall of Auto-SCAT for activated relevant checks per project with threshold equals to 0.3.

Project name	**Precision**	**Recall**
Couchbase	62.5%	71.4%
Eclipse CDT	73.0%	75.0%
Eclipse JDT Core	100.0%	75.0%
Eclipse Platform UI	62.9%	75.9%
OpenDaylight Controller	88.9%	71.4%
Vaadin	48.6%	64.2%
OVERALL (Mean)	**72.7%**	**72.2%**

Table 5 Macro-averaged Precision and Recall of Auto-SCAT for activated relevant categories per project for threshold 0.3.

Project name	Precision	Recall
Couchbase	100.0%	100.0%
Eclipse CDT	90.0%	100.0%
Eclipse JDT Core	100.0%	100.0%
Eclipse Platform UI	87.5%	87.5%
OpenDaylight Controller	100.0%	90.9%
Vaadin	100.0%	100.0%
OVERALL (Mean)	**96.3%**	**96.4%**

5.2.2 $RQ_{2.2}$: How accurate is Auto-SCAT to configure CheckStyle?

Table 4 shows the performance of Auto-SCAT regarding the enabling of relevant checks for each project. It is possible to note that Auto-SCAT enables relevant checks with a macro-averaged precision varying from 48.6% up to 100%, and a macro-averaged recall varying in the range [64.2%-75.9%]. Specifically, Auto-SCAT shows an overall macro-averaged precision of 72.7% and an overall macro-averaged recall of 72.2%.

While moving the attention on the accuracy of Auto-SCAT in enabling relevant categories of CheckStyle for the studied projects, as reported in Table 5, we observe a very high precision and recall, 96.3% and 96.4%, respectively. Since in this case Auto-SCAT only recommends a category and not a specific check, both precision and recall are better, at the cost of a coarse-grained recommendation.

Table 6 Auto-SCAT and baselines compared on Mylyn's code review comments and coding guidelines.

	Check-level		**Category-level**	
Configuration	**Micro Prec.**	**Micro Recall**	**Micro Prec.**	**Micro Recall**
Auto-SCAT (@0.3; comments)	89.7%	89.7%	91.1%	91.1%
Auto-SCAT (@0.4; comments)	93.8%	93.8%	94.5%	94.5%
Auto-SCAT (@0.3; guidelines)	30.0%	45.0%	66.7%	85.7%
Auto-SCAT (@0.4; guidelines)	33.3%	25.0%	83.3%	71.4%
Sun/Oracle (default)	21.3%	65.0%	54.5%	85.7%
Google (default)	24.1%	65.0%	54.5%	85.7%
Historical	15.8%	60.0%	46.2%	85.7%

RQ_2 Summary: Auto-SCAT leverages code review comments to configure CheckStyle checks with micro- and macro-averaged precision of $\simeq$ 75% and 32% (see Fig. 3), and CheckStyle categories with micro- and macro-averaged precision of 79.6% and 46.8% (see Table 3) for an optimal check calibration. Moreover, Auto-SCAT shows consistent results across projects when a default threshold is used. Finally, Auto-SCAT enables checks with high macro-averaged precision and recall (72.7% and 72.2%, respectively, see Table 4) and enables almost all the relevant categories (96.3% macro-averaged precision and 96.4% macro-averaged recall, see Table 5).

5.3 *RQ_3: How accurate is Auto-SCAT compared to a baseline?*

Table 6 reports the results of comparing the configuration generated by Auto-SCAT, at check and category-level with thresholds 0.3 and 0.4, to the baselines when using the Mylyn's code review comments and Mylyn's coding guidelines.

As also done for RQ_2, we have evaluated the performance of Auto-SCAT using knowledge bases constructed differently, *i.e.,* combining check descriptions and code reviewers comments in different ways, and report the one that performs better, *i.e.,* in this case the one that also considers the checks' description provided by the CheckStyle configuration in case no code review comments are belonging to them, *i.e.,* Comments + descriptions, or just descriptions. We conjecture that, in a practical application scenario, the different approaches could be tested, or the ones involving descriptions could be preferred especially when there is little or no history available.

As the table shows, Auto-SCAT classifies code review comments with a high precision and a high recall at both check and category-level (89.7% and 91.1%, respectively).

Unsurprisingly, precision and recall of the Sun/Oracle and Google CheckStyle default configurations are also much lower than Auto-SCAT. This can be explained because (i) default configurations enable all possible CheckStyle checks, many of which may not be relevant for the studied projects' coding standards (this explains the low precision). Moreover, the thresholds for each check are, again, default ones and not configured on projects' preferences. For this reason, also the recall is somewhat low, although not as low as the precision (*i.e.,* $\simeq$65% at check-level and $\simeq$86% at category-level).

We need to remark that, as already stated in Section 4, our goal is to reduce the number of irrelevant warnings produced by SCATs that hinder their adoption in software development process (Beller et al., 2016; Johnson et al., 2013; Wedyan et al., 2009; Zampetti et al., 2017) in order to increase their usefulness, *i.e.,* increase the precision of Auto-SCAT. To quantify the usefulness of Auto-SCAT, we estimate the time that it would save developers. Specifically, we estimate the extra time that developers would spend analyzing irrelevant warnings triggered by the historical configurations by identifying the false positive warnings of these configurations that are not present in the configuration generated by Auto-SCAT. Results show that the historical approach generates 10,253 extra irrelevant warnings not generated by Auto-SCAT. Considering that it takes approximately 8 minutes to analyze a warning (Ruthruff et al., 2008), Auto-SCAT saves about 8.5 months of a developer working full time.

RQ$_3$ Summary: Auto-SCAT outperforms the simple use of a default CheckStyle configuration, but also the use of historical data to configure the SCAT both at check and category level when compared with the Mylyn's coding guidelines as well as in terms of performance of the classification of code review comments.

6 Threats to validity

Threats to *construct validity* concern the relationship between theory and observation. One important threat can be represented by the ground-truth used in our study. Such a ground-truth consists of a manually-produced mapping between code review comments and CheckStyle checks. In particular, (i) we could have imprecision due to the manual mapping, and (ii) it is possible that, while a suggestion is given in a code review, it might be never followed up in the project. To mitigate this threat we have involved two annotators inspecting each comment-warning traceability link, and in RQ$_3$ we have assessed our results with respect to a ground-truth consisting of Mylyn's coding standards. Moreover, while our approach is able to discard comments that do not deal with stylistic issues with an overall precision of 82.6% and a recall of 90.0%, Auto-SCAT achieves less positive results in identifying the proper CheckStyle check (precision of 60.7% and a recall of 51.6%). This result is mainly caused by the few data points in our gold standard for specific checks. We plan to address this issue in the future by targeting the addition of more data points to the less represented categories.

Threats to *internal validity* concern factors, internal to our study, that could have influenced the results. A major factor that could impact the performance of Auto-SCAT is represented by its various settings. However, we show how results vary for different similarity thresholds. Also, we have experimented different text preprocessing steps and different ways of building the knowledge base and used the one achieving the best performance.

Threats to *external validity* are related to the generalizability of the results, as the study is limited to CheckStyle and to six Java open source projects from four different ecosystems (Couchbase, Eclipse, Vaadin and Open Daylight). While our empirical evaluation only shows how well CheckStyle can be configured using code review comments, many SCATs detect similar issues such as documentation, style, and program design. Therefore, a similar approach could be applied to other SCATs.

7 Conclusion and Future Work

This paper proposes Auto-SCAT, a novel approach to automatically configure Static Code Analysis Tools (SCATs), by leveraging the content of code review comments. We evaluated Auto-SCAT on data from six Java open source projects, to automatically configure CheckStyle. Overall, Auto-SCAT is able to correctly map code review comments onto CheckStyle checks with a precision of 75.1% and a recall of 74.6%. Also, based on the evaluation of Auto-SCAT on data from a project (Mylyn) for which coding standards are available, we show that Auto-SCAT outperforms default CheckStyle configurations and a baseline leveraging historical data.

Future work is aimed at extending the approach to further SCATs, and to perform a fine-grained configuration of the checks, *i.e.,* to automatically configure check's parameters. As discussed in Section 5.2.1, the Auto-SCAT accuracy can depend on the vocabulary consistency between code reviews and check descriptions, as well as on the extent to which there is a vocabulary drift between the corpus used for training and the one on which Auto-SCAT is used. To this extent, it could be useful to investigate the use of thesaurus or lexical databases to cope with such a vocabulary mismatch. Last, but not least, to enhance the completeness of SCAT configurations captured by Auto-SCAT, we will analyze data sources beyond code reviews, *e.g.,* issues, emails or chat logs, to investigate which of these sources can be used to improve results of the proposed approach.

References

Anderson P, Reps T, Teitelbaum T, Zarins M (2003) Tool support for fine-grained software inspection. In: IEEE Software

Ayewah N, Pugh W (2009) Using checklists to review static analysis warnings. In: Proceedings of the International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009), pp 11–15

Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: Proceedings of the International Conference on Software Engineering (ICSE), pp 712–721

Baeza-Yates RA, Ribeiro-Neto B (1999) Modern Information Retrieval. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Bavota G, Russo B (2015) Four eyes are better than two: On the impact of code reviews on software quality. In: IEEE International Conference on Software Maintenance and Evolution, (ICSME)

Baysal O, Kononenko O, Holmes R, Godfrey M (2013) The influence of non-technical factors on code review. In: Reverse Engineering (WCRE), 2013 20th Working Conference on

Beller M, Bacchelli A, Zaidman A, Juergens E (2014) Modern code reviews in open-source projects: Which problems do they fix? In: Proceedings of the Working Conference on Mining Software Repositories (MSR), pp 202–211

Beller M, Bholanath R, McIntosh S, Zaidman A (2016) Analyzing the state of static analysis: A large-scale evaluation in open source software. In: Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp 470–481

Bosu A (2014) Characteristics of the vulnerable code changes identified through peer code review. In: 36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014, pp 736–738

Bosu A, Carver JC, Bird C, Orbeck J, Chockley C (2017) Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. IEEE Transactions on Software Engineering

Cassee N, Vasilescu B, Serebrenik A (2020) The silent helper: The impact of continuous integration on code reviews. In: 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020, pp 423–434

Couto C, Montandon JaE, Silva C, Valente MT (2013) Static correspondence and correlation between field defects and warnings reported by a bug finding tool. Software Quality Journal 21(2):241–257

D Zhang YG D Jin, Zhang H (2013) Diagnosis-oriented alarm correlations. In: Asia-Pacific Software Engineering Conference (APSEC)

Duvall P, Matyas SM, Glover A (2007) Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series). Addison-Wesley Professional

Fagan M (1976) Design and code inspections to reduce errors in program development. IBM Systems Journal

Fry Z, Weimer W (2013) Clustering static analysis defect reports to reduce maintenance costs. In: Proceedings of the Working Conference on Reverse Engineering (WCRE)

Hanam Q, Tan L, Holmes R, Lam P (2014) Finding patterns in static analysis alerts: Improving actionable alert ranking. In: Proceedings of the Working Conference on Mining Software Repositories

Huang A (2008) Similarity measures for text document clustering. In: New Zealand Computer Science Research Student Conference

Johnson B, Song Y, Murphy-Hill E, Bowdidge R (2013) Why don't software developers use static analysis tools to find bugs? In: Proceedings of the International Conference on Software Engineering (ICSE), pp 672–681

Khoo YP, Foster JS, Hicks M, Sazawal V (2008) Path projection for user-centered static analysis tools. In: Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp 57–63

Kim S, Ernst M (2007) Which warnings should I fix first? In: Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pp 45–54

Kononenko O, Baysal O, Godfrey MW (2016) Code review quality: How developers see it. In: Proceedings of the 38th International Conference on Software Engineering

Manning CD, Raghavan P, Schütze H (2008) Introduction to Information Retrieval. Cambridge University Press

Mäntylä M, Lassenius C (2009) What types of defects are really discovered in code reviews? IEEE Trans Software Eng 35(3):430–448

Marcilio D, Bonifacio R, Monteiro E, Canedo E, Luz W, Pinto G (2019) Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In: International Conference on Program Comprehension(ICPC)

McIntosh S, Kamei Y, Adams B, Hassan AE (2016) An empirical study of the impact of modern code review practices on software quality. Empirical Software Engineering 21(5):2146–2189

Morales R, McIntosh S, Khomh F (2015) Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In: Proc. of the 22nd Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER)

Muske T, Baid A, Sanas T (2013) Review efforts reduction by partitioning of static analysis warnings. In: Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)

P Cousot, R Cousot, J Feret, L Mauborgne, and D Monniaux A, , X Rival (2005) The astreé analyzer. In: Proceedings of the European Symposium on Programming (ESOP)

Panichella S, Zaugg N (2020) An empirical investigation of relevant changes and automation needs in modern code review. Empirical Software Engineering 25(6):4833–4872

Panichella S, Arnaoudova V, Di Penta M, Antoniol G (2015) Would static analysis tools help developers with code reviews? In: Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp 161–170

Pascarella L, Spadini D, Palomba F, Bruntink M, Bacchelli A (2018) Information needs in contemporary code review. PACMHCI 2(CSCW):135:1–135:27

Phang K, Foster JS, Hicks MW, Sazawal V (2009) Triaging checklists: a substitute for a phd in static analysis. Evaluation and Usability of Programming Languages and Tools (PLATEAU)

Porter M (1980) An algorithm for suffix stripping. Program 14(3):130–137, DOI 10.1108/eb046814

Querel LP, Rigby PC (2018) Warningsguru: integrating statistical bug models with static analysis to provide timely and specific bug warnings. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 892–895

Řehůřek R, Sojka P (2010) Software Framework for Topic Modelling with Large Corpora. In: Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks, ELRA, pp 45–50

Reiss S (2007) Automatic code stylizing. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp 74–83

Ribeiro A, Meirelles P, Lago N, Kon F (2019) Ranking warnings from multiple source code static analyzers via ensemble learning. In: Proceedings of the 15th International Symposium on Open Collaboration, ACM, p 5

Rigby PC, German DM, Storey MA (2008) Open source software peer review practices: A case study of the apache server. In: Proceedings of the 30th International Conference on Software Engineering

Ruthruff JR, Penix J, Morgenthaler JD, Elbaum S, Rothermel G (2008) Predicting accurate and actionable static analysis warnings: an experimental approach. In: Proceedings of the International Conference on Software Engineering (ICSE), pp 341–350

Salton GM, Wong A, Yang C (1975) A vector space model for automatic indexing

Sokolova M, Guy L (2009) A systematic analysis of performance measures for classification tasks. Information Processing & Management pp 427–437

Spacco J, Hovemeyer D, Pugh W (2006) Tracking defect warnings across versions. In: Proceedings of the 2006 international workshop on Mining software repositories, ACM, pp 133–136

Vassallo C, Panichella S, Palomba F, Proksch S, Zaidman A, Gall HC (2018) Context is king: The developer perspective on the usage of static analysis tools. In: Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 38–49

Wedyan F, Alrmuny D, Bieman JM (2009) The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In: Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009, IEEE Computer Society, pp 141–150

Weißgerber P, Neu D, Diehl S (2008) Small patches get in! In: Proceedings of the 2008 International Working Conference on Mining Software Repositories

Williams CC, Hollingsworth JK (2005) Automatic mining of source code repositories to improve bug finding techniques. IEEE Transactions on Software Engineering

Yang Y (1999) An evaluation of statistical approaches to text categorization. Information Retrieval 1(1):69–90

Yoon J, Jin M, Jung Y (2014) Reducing false alarms from an industrial-strength static analyzer by SVM. In: 21st Asia-Pacific Software Engineering Conference, APSEC 2014, Jeju, South Korea, December 1-4, 2014. Volume 2: Industry, Short, and QuASoQ Papers, pp 3–6

Yüksel U, Sözer H (2013) Automated classification of static code analysis alerts: A case study. In: Proceedings of the International Conference on Software Maintenance

Zampetti F, Scalabrino S, Oliveto R, Canfora G, Di Penta M (2017) How open source projects use static code analysis tools in continuous integration pipelines. In: Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017, pp 334–344

Zampetti F, Mudbhari S, Arnaoudova V, Di Penta M, Panichella S, Antoniol G (2020) Using Code Reviews to Automatically Configure Static Analysis Tools. DOI 10.5281/zenodo.4399225, URL `https://doi.org/10.5281/zenodo.4399225`

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