EvaCRC: Evaluating Code Review Comments

Lanxin Yang
State Key Laboratory of Novel Software Technology, Software Institute, Nanjing University
Nanjing, China
lanxin.yang@smail.nju.edu.cn

Jinwei Xu
State Key Laboratory of Novel Software Technology, Software Institute, Nanjing University
Nanjing, China
jinwei_xu@smail.nju.edu.cn

Yifan Zhang
State Key Laboratory of Novel Software Technology, Software Institute, Nanjing University
Nanjing, China
yifan.zhang@smail.nju.edu.cn

He Zhang
State Key Laboratory of Novel Software Technology, Software Institute, Nanjing University
Nanjing, China
hezhang@nju.edu.cn

Alberto Bacchelli
Department of Informatics
University of Zurich
Zurich, Switzerland
bacchelli@ifi.uzh.ch

ABSTRACT

In code reviews, developers examine code changes authored by peers and provide feedback through comments. Despite the importance of these comments, no accepted approach currently exists for assessing their quality. Therefore, this study has two main objectives: (1) to devise a conceptual model for an explainable evaluation of review comment quality, and (2) to develop models for the automated evaluation of comments according to the conceptual model. To do so, we conduct mixed-method studies and propose a new approach: **EvaCRC** (**Eva**luating **C**ode **R**eview **C**omments). To achieve the first goal, we collect and synthesize quality attributes of review comments, by triangulating data from both authoritative documentation on code review standards and academic literature. We then validate these attributes using real-world instances. Finally, we establish mappings between quality attributes and grades by inquiring domain experts, thus defining our final explainable conceptual model. To achieve the second goal, EvaCRC leverages multi-label learning. To evaluate and refine EvaCRC, we conduct an industrial case study with a global ICT enterprise. The results indicate that EvaCRC can effectively evaluate review comments while offering reasons for the grades.
Data and materials: https://doi.org/10.5281/zenodo.8297481

CCS CONCEPTS

• **Software and its engineering → Software development process management**; **Empirical software validation**.

KEYWORDS

Code review, review comments, quality evaluation

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616245

ACM Reference Format:
Lanxin Yang, Jinwei Xu, Yifan Zhang, He Zhang, and Alberto Bacchelli. 2023. EvaCRC: Evaluating Code Review Comments. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616245

1 INTRODUCTION

Code review was designed primarily to detect quality issues in software development [26], then was found useful in identifying code weaknesses and seeking opportunities for improvement in design, testing, and security (*e.g.*, [7, 51, 64, 81]). Apart from supporting quality assurance, code review benefits other aspects such as team awareness [2], knowledge transfer [10], and tool improvement [80]. Compared with other quality assurance practices (*e.g.*, testing), code review more largely relies on organizational culture and developers' expertise, experience, participation, and engagement [3–5, 41, 72]. Moreover, code review sometimes may become wasteful or even harmful for projects and teams due to its confusion [22], time- and effort-consuming [36, 59], low efficiency and effectiveness [15, 33, 35, 42], and other negative side-effects [11, 21, 75].

Over the past few years, standards, guidelines, and experience have been shared to improve code review (*e.g.*, IEEE Computer Society [62], Google [30, 60], and GitLab [29]). Most of these resources underline the key role of review feedback, as shared through review comments. Review comments serve as a communication tool for reviewers to share their assessment of code changes. Thus, these comments are the key guide for corrections and improvements.

Despite the importance of review comments, however, only few studies focus on their evaluation. Two studies [6, 34] have investigated developers' perception of the usefulness of review comments and developed models for automating the evaluation process. However, these approaches output a binary label (*i.e.*, useful or not), without providing any additional information to justify it or suggest improvements. This binary assessment could inadvertently overlook potentially 'useful' review comments, potentially leading to developers contesting the results or even resisting the evaluation. Existing predictive models predominantly depend on *review context features* (*e.g.*, *change_trigger* [6]: determining if any changes occur

within a line of the highlighted comment in subsequent iterations, and *gratitude* [34]: assessing if the code author responds with expressions like 'Thank you'). They also utilize *experience features* (*e.g.*, *code_authorship* [57]: counting the commits a developer has made on a file and *external_library_experience* [57]: gauging a developer's familiarity with external libraries from a reviewed file). A significant limitation of these features is their availability only *after* the completion of the review, thus falling short in offering instantaneous feedback to the author of the comments.

We contend that the evaluation of review comments should go beyond assigning a grade. Instead, the evaluation should guide corrections or enhancements. Herein, we propose an approach, **EvaCRC** (**Eva**luating **C**ode **R**eview **C**omments), to support effective evaluation of review comments. EvaCRC combines a conceptual model and an automated model. The former aims to offer an explainable conceptual model for review comments, while the latter aims to automate their evaluation. Following the guidelines for mixed-method research design [13], we devise two studies for implementing and evaluating EvaCRC, respectively.

In the first study, to implement the conceptual model of EvaCRC, we adopt data triangulation (including authoritative documents, academic literature, and real-world examples) to collect, synthesize, and validate quality attributes. Our resulting conceptual model consists of four review context-independent quality attributes that characterize review comments (*i.e.*, '*emotion*', '*question*', '*evaluation*', and '*suggestion*') and their mappings to one of four quality grades (*i.e.*, '*excellent*', '*good*', '*acceptable*', and '*poor*'). As opposed to prior approaches (*e.g.*, [34, 57]), which output only a binary grade (*i.e.*, useful or not), EvaCRC is designed to output one overall quality grade together with four specific quality attributes that explain it. For instance, given the review comment– "*getAbsolutePath() is not allowed in security scanning, use getCanonicalPath() instead.*" EvaCRC evaluates it as 'acceptable', with the quality attributes values being 'emotion'–Positive (1), 'question'–No (0), 'evaluation'–Yes (1), and 'suggestion'–Yes (1), respectively. According to the conceptual model, EvaCRC generates the overall quality grade for each review comment by following the predefined mappings (cf. Table 3) as evaluated and approved by developers at a large Information and Communication Technology (ICT) enterprise, and, the four specific quality attributes are learned from genuine review comments. We formulate the learning task as multi-label text classification. Since manual classification is time-consuming and bias-prone, we use natural language processing and machine learning to develop text classifiers to form an automated model of EvaCRC.

In the second study, to evaluate and improve EvaCRC, we carry out an industrial case study at a large ICT enterprise. We first continue to implement the conceptual model by establishing the enterprise-specific mappings between quality attributes and grades. We then collect and annotate 17,000 review comments and build six multi-label text classifiers to seek the most effective for forming the automated model of EvaCRC. Finally, we interview software practitioners at this ICT enterprise to understand their perception of EvaCRC. The results from the case study indicate that EvaCRC can effectively evaluate review comments, providing reviewers not only evaluation results (quality grades) that are more acceptable than a binary evaluation but detailed explanations (quality attributes) for timely correction and improvement.

2 RELATED WORK

This section reviews the literature that is relevant to our work.

Usefulness of Review Comments. The only aspect concerning the quality of review comments investigated so far is *usefulness*.

Bosu et al. [6] interviewed seven developers at Microsoft, discussing with them a total of almost 150 comments in terms of usefulness. Developers defined as '*useful*' review comments that (1) indicate functional issues; (2) identify validation issues, alternate scenarios; and (3) transfer knowledge. Whereas, developers rated as '*somewhat useful*' review comments that indicate: (1) nitpicking issues; (2) indentation, comments, style, identifier naming issues; and (3) alternatives. Finally, developers rated as '*not useful*' comments that merely present (1) questions, (2) appreciation, and (3) suggestions for future improvement.

Hasan et al. [34] interviewed Samsung's developers and found that they define as '*useful*' review comments that identify at least one of the following: (1) defects, (2) missing input validations, (3) opportunities for making code efficient and optimized, (4) readability weakness, (5) logical mistakes, (6) redundant code, (7) corner cases, (8) opportunities for integrating code, (9) deprecated functions, (10) design weakness, and (11) coding standard violations. A review comment is '*not useful*' when it regards: (1) visual representation issues that can be also identified by static analysis tool, (2) false positive, (3) misinterpretation, (4) discussion on an already resolved issue, or (5) solution approach with which the author disagrees. On some types of comments there was no consensus: praise, clarification questions, and suggestions on improving documentation.

Automated Evaluation of Review Comments. Bosu et al. [6] created an automated approach to classify review comments as either '*useful*' (also including their '*somewhat useful*' category) or '*not useful*'. Their approach was developed by annotating 989 examples and devising a decision tree with hand-crafted rules. Their approach heavily relies on the impact of comments, particularly whether it triggered a subsequent change or whether it was labeled as '*wontfix*'. Their model reached mean precision, recall, and accuracy of 89%, 85%, and 83%, respectively.

Unlike Bosu et al. [6], Rahman et al. [57] only leveraged "change triggering" to label a dataset of the usefulness of review comments. Rahman et al. computed features concerning both the 'textual review comments' (*e.g.*, *reading ease* and *code element ratio*) and the 'reviewer experience' (*e.g.*, *code authorship* and *external library experience*) for developing prediction models. The authors collected 1,482 annotated examples and developed three learning-based classifiers: Naïve Bayes, Logistic Regression, and Random Forests for binary evaluation. Random Forests achieved the best performance with a mean precision of 65.76%, mean recall of 65.89%, mean F1-score of 65.82%, and mean accuracy of 66.06%.

In addition to adopting the features presented in Bosu et al. [6] and Rahman et al. [57], Hasan et al. [34] devised more features from the perspectives of 'textual review comments' (*e.g.*, *word count*), 'review contexts' (*e.g.*, *author responded* and *reply sentiment*), and 'reviewer experience' (*e.g.*, *reviewing_experience*). They collected 2,004 annotated examples and trained six classifiers: Decision Tree, Random Forests, Support Vector Machine, Multi-Layer Perceptron, XGBoost, and Logistic Regression. The best-performing classifier is Random Forests with a mean accuracy of 87.32%.

Overall, the studies have created models to automatically predict the usefulness of review comments without providing any explanation. However, the unexplained results may lead to developers' disagreements with the results or even resistance to the evaluation itself. In addition, the prediction models rely on a large number of complex hand-crafted features some of which can only be assessed after the comments have been read and acted upon by other developers (*e.g.*, whether the comment triggered a change or whether it has been labeled as '*wontfix*'). Therefore, the proposed approaches have limited to no use in practice, for example, to give immediate feedback to the author of a review comment about what they have written. Furthermore, the heavy feature engineering work (*e.g.*, collecting and calculating complex features from multi-source heterogeneous data, selecting suitable features, and scaling features) asks for a great amount of time and effort.

3 STUDY OVERVIEW

We start by presenting real-world examples of review comments (cf. Table 1), which we use to motivate why it is important and challenging to evaluate review comments. Then we present an overview of our proposed approach, followed by our research questions.

Table 1: Real-world examples of code review comments

id	Comment
C1	"delete"
C2	"magic number"
C3	"Why do you want this?"
C4	"why again hardcoding???"
C5	"I think it should be "Logger.info"
C6	"should it be public or protected?"
C7	"change the logging level to debug."
C8	"downloadTips → mDownloadTips"
C9	"getAbsolutePath() is not allowed in security scanning, use getCanonicalPath() instead."
C10	"Please discuss box name with @xYZ1111111, we have a box naming rule and use it in tracing data."
C11	"Check whether these scripts already executed or need to execute in 3.17.4.103 and place the scripts accordingly."
C12	"Do these values refer to specific values? If you don't need to use minus-separated statements like 'FULL-TIME', remove the minuses and quotes. Otherwise, keep just the minus-separated statements and remove quotes from the others."

3.1 Evaluating Review Comments: Challenges

Several challenges arise when evaluating review comments.

Challenge 1: Disagreements by Reviewers. Past research (*e.g.*, [2, 47]) has recognized that there is a significant variance in the quality maturity among review comments. How can these differences be characterized and graded? There are no standard criteria and approaches for evaluating review comments.

In our industrial case study (cf. Section 5), one manager we interviewed explained that it is challenging to generally evaluate the *usefulness* of a comment, especially on a binary scale; as she put it: "*It is hard to say that a review comment is absolutely not useful in practice . . . even one interrogation mark can serve as a reminder to developers. However, we are seeking to develop guidelines and standards for performing code reviews as well as evaluation criteria. Otherwise, code review becomes wasteful.*" Other developers argued that using only a *usefulness* criterion is more likely to result in biases and carelessness. To define the quality of a review comment, some developers suggested taking into account the type of defects it detects (*e.g.*, comments detecting security or critical issues are to be considered high-quality).

Finally, providing negative feedback to reviewers about their comments may result in disagreements and negative feelings, especially if no explanation is provided. Negative evaluations may impact reviewers' productivity in either code reviews or/and other software activities, possibly leading them to leave project teams in extreme cases.

Challenge 2: Modeling Review Comments. Review comments often include jargons, abbreviations, and misspellings. Table 1 presents real-world examples to illustrate the linguistic *diversity and complexity* of review comments: (1) *lexicon:* natural language, programming language, and special symbols are mixed (C5, C6, C8, C9); (2) *syntax:* declarative (C5), imperative (C7), general interrogative (C3), and selective interrogative (C6); and (3) *pragmatics:* indication (C2), question (C4, C11), suggestion (C1, C12), and assignment (C10). In addition, review comments transfer reviewers' emotions (*e.g.*, C4), which have impacts on developers' feelings as well as on the quality of the project they are working on [24, 25, 48, 49].

Moreover, review comments transfer complex functional semantics of the language (*e.g.*, causality, contrast, exemplification, clarification, similarity, and hypothesis [23]), and offer multiple practical functions (*e.g.*, correctness, decision, management, and interaction [45]). When it comes to the primary goal of code review (*i.e.*, finding defects[1] [2, 59, 60]), review comments indicate multiple types, such as evolvability or functional defects, and compiler errors [56]. Gunawardena et al. [31] identified 116 types of defects that fell into 15 groups in code review. The diversity and complexity of review comments make them hard to comprehend and model.

3.2 Evaluating Review Comments: Goals

We strive to achieve the following goals in our studies.

Goal 1: Explainable Evaluation. We aim to establish an explainable conceptual model. The term 'explainable' indicates that the conceptual model should output not only the final quality grade (as previous studies did), but also the causes for that grade. We place emphasis on letting developers know the causes behind evaluations to help them learn and improve. We also strive to ensure that the conceptual model is independent of the review context (*e.g.*, author/reviewership, review process&outcome). Let us consider a review comment that is merely a question mark. In one review, it may be evaluated as '*useful*' because it triggers code changes; whereas, in another review, it is evaluated as '*not useful*' because it triggers no code changes. Such an evaluation, only based on the effect of the comment, offers little help to improve. Instead, the evaluation should offer explanations thus leading to improvements.

Goal 2: Automated Evaluation. Automating the process of evaluation is essential due to the time- and effort-consuming nature of the manual evaluation of comments. We aim to develop a model that automatically learns the high-level representation of review comments to tackle their linguistic complexity. We aim to avoid heavy hand-crafted feature engineering (*e.g.*, feature extraction and

[1] In this paper, the term 'defect' is used as a general concept that depicts any condition deviating from expectations based on predefined standards, or from developers' experiences. The 'defect' could be 'bug', 'vulnerability', 'error', etc., in a specific context.

selection) for training automated models. Therefore, we investigate natural language processing and deep learning for automation.

We expect the conceptual model (quality attributes and mappings) rather than the automated models to provide an explanation of the evaluation results, as researching the latter (*e.g.*, explaining how the neural network-based models understand semantics from genuine review comments) is beyond the scope of this study.

3.3 Evaluating Review Comments: Approach

Our approach (**EvaCRC**) is a synergy of two sub-goals: *explainable evaluation* and *automated evaluation*. The ultimate goal **G** (to associate two sub-goals) of this study depends on the function **F** (to learn quality attributes **Q** from review comments) and the mapping **Z** (to determine quality grades with attributes). Once the attributes are determined and their mappings to grades are established, the remaining task is to seek appropriate models and strategies for learning the objective function **F**. Our goal can be formulated as a supervised multi-label text classification problem.

Figure 1 shows an overview of research designs of this mixed-method research. In the first study, we implement the designs (as presented above) of EvaCRC; in the second study, we evaluate EvaCRC as well as seek improvements.

3.4 Research Questions

Our study is structured around three Research Questions (RQs).

RQ1: What quality attributes are appropriate for the conceptual model?
Motivation: We propose RQ1 to guide the collection, synthesis, and validation of quality attributes to form the core components of the conceptual model.
Methods: We address RQ1 by using *data triangulation* [18] and *thematic synthesis* [14]. As shown in Figure 1 (the top of the *Implementation* part), our triangulated data sources include authoritative documents, academic literature, and real-world examples. Triangulation is the high-level method we used, while thematic synthesis is used to synthesize quality attributes, specifically.

RQ2: How effective are text classifiers in predicting (RQ2.1) the quality attributes and (RQ2.2) the quality grades?
Motivation: As little is known about how well text classifiers perform, we propose RQ2 to guide the comparisons.
Method: To address RQ2, we conduct *experiments* [73] (the bottom of the *Implementation* part in Figure 1) in which we use the review comments collected from an ICT enterprise as the dataset, we set six multi-label text classifiers and evaluate them using four metrics for answering RQ2.1, and three metrics for answering RQ2.2.

RQ3: How do practitioners perceive EvaCRC?
Motivation: We investigate how practitioners perceive the effectiveness of EvaCRC as a further evaluation and to guide future improvements.
Method: To address RQ3, we conduct *focus group interviews* [55] (the *Evaluation* part in Figure 1) with practitioners from the ICT enterprise that provides us with the experimental dataset. So that we could receive insightful comments and recommendations.

4 STUDY I: IMPLEMENTING EVACRC

We describe our mixed-method study for implementing EvaCRC.

4.1 Conceptual Model

4.1.1 Quality Attributes. Our original intention was to collect quality attributes from Fagan's article [26] and IEEE Computer Society's standards [62], which are the main *authoritative documents* concerning code review. Unfortunately, these did not directly meet our goal as expected. Therefore, we followed the guidelines [61] to adopt data triangulation (*i.e.*, a "combination of methodologies in the study of the same phenomenon" [19]) to gather different types of evidence to collect quality attributes and cross-validate them. Data triangulation increases the confidence of research data, creates innovative and multi-perspective ways of understanding a problem, and reveals unique findings [37].

With data triangulation [66], we added data sources (*i.e.*, *academic literature* and *real-world examples*) and tailored our data collection methods based on the best fit with the goal [9]. The research method per data source is elaborated as follows.

Source 1: Authoritative Documents. We investigated (1) "*Design and code inspections to reduce errors in program development*" (Michael Fagan, 1976) [26], and (2) "*IEEE standard for software reviews and audits*" (IEEE Computer Society, 2008) [62].

The former is the seminal study that formalizes *code inspection* (a.k.a. classical code review), while the latter presents standards for software reviews and audits. Even though several types of reviews exist (*e.g.*, management reviews, technical reviews, and walkthroughs), they are all required to review source code, *i.e.*, code review is fundamental [77]. From these documents, we found that the core objective of code review is reducing defects—this is the primary quality attribute ('reducing defects').

Source 2: Academic Literature. The first additional data source is academic literature. Since academic literature has been peer-reviewed, the rigor and credibility of its contents can be guaranteed to some extent. Although 'code review comments' are the subject of our research, our initial searches with Google Scholar, IEEE Xplore Digital Library, and ACM Digital Library indicated that limited literature refers specifically to review comments. Therefore, to gather our quality attributes, we broadened our search scope and analyzed literature whose subject is "code review".

We employed *rapid literature review* [67]—a form of evidence collection and synthesis that simplifies components of a systematic (literature) review to quickly acquire knowledge [67]. To retrieve the primary studies to analyze, we gathered the papers from the two most up-to-date, code review-related systematic reviews (*i.e.*, [16, 71]). Both reviews were published in 2021 in the *Journal of Systems and Software*. Davila and Nunes [16] collected 139 primary studies from 2005 to 2019. Wang et al. [71] collected 112 primary studies published between 2011 and 2019. By combining the two sets, we were left with 182 primary studies to analyze.

Since only a few studies directly present quality attributes, we extracted two items (**Benefits** and **Challenges**) from each primary study (if they exist) to synthesize quality attributes. Both items are associated with code review, working for investigating (1) *what benefits can be fully achieved* and (2) *what challenges can be to some extent mitigated by evaluating and improving the quality of review*

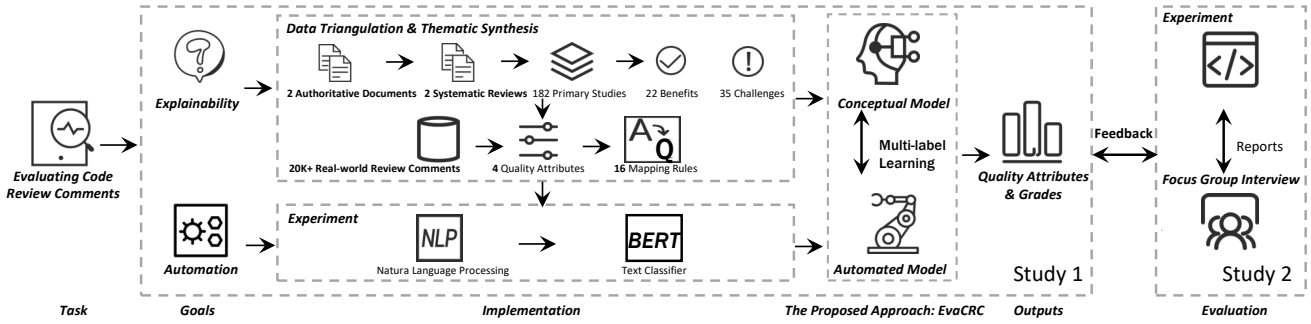

Figure 1: An overview of mixed-method research designs (with several results)

comments. At the same time, with reference to the *authoritative documents*, the core objective of code review is extracted to further guide the attribute synthesis. Moreover, we used another new data source (*i.e.*, *real-world examples*) to validate and improve the quality attributes output in this study. The research methods and procedures are as follows.

We synthesized quality attributes from the two sets of items by using *thematic synthesis* [14]. According to Cruzes and Dybå [14], a general thematic synthesis requires steps to: (1) extract data (S_1), (2) code data (S_2), (3) translate codes into themes (S_3), (4) create a model of higher-order themes (S_4), and (5) assess the synthesis' trustworthiness (S_5).

In S_1, we assigned the sample set to two researchers with prior experience with analyzing qualitative data. Their goal was to extract *B*enefits and *C*hallenges by reading each primary study's abstract, introduction, and any other sections if needed. Because the abstract and introduction may outline the code review-related benefits and challenges, while the others focus on the specific subject and technical issues (*e.g.*, improving the accuracy of recommending reviewers [50]). Both researchers started by analyzing 20 studies together to coordinate on the task, then independently analyzed the remaining studies.

In S_2, those two researchers performed *open coding* with existing keywords in the text. For instance, given the primary study [53] stating: "*Contemporary code review is a widespread practice used by software engineers to maintain high software quality and share project knowledge*" the researchers extracted '*software quality assurance*' and '*knowledge exchange*' as *B*enefits, and for the sentence: "*However, conducting proper code review takes time and developers often have limited time for review*" the researchers extracted '*time-consuming*' as *C*hallenge. We did not measure the inter-coder reliability at this stage due to the lack of predefined, complete items before performing coding. Instead, the researchers verified together the extraction results and made any corrections after reaching an agreement. The extraction results were also randomly checked by another researcher to further mitigate biases. Through this step, we summarized 22 benefits and 35 challenges. The most frequently mentioned *benefits* are: (1) defect detection&fix (B_1), (2) software (code) quality assurance&improvement (B_2), (3) knowledge transfer (B_3), (4) increasing team awareness (B_4), and (5) exploring alternative solutions (B_5), etc. The most frequently mentioned *challenges* include: (1) finding suitable reviewers (C_1), (2) time-&cost-&effort-consuming (C_2), (3) understanding code changes (C_3), (4) fairness/bias/conflict (C_4), (5) defect escaping (C_5), (6) increasing workload (C_6), (7) lack of support tools (C_7), and (8) expertise needed (C_8), etc. (the complete data items are available in the online replication package [76].)

In S_3, we excluded the *benefits* and *challenges* that cannot be analyzed from the review comments (*e.g.*, C_1, C_2). For instance, we removed C_2 because EvaCRC is expected to be review context-independent (*i.e.*, the only data source/input is textual review comments). Although the time, effort, and cost may impact the quality of review comments, they would also change with the knowledge and experience of reviewers and the complexity of code changes reviewed (which are difficult to measure and can be affected by biases). Then, we performed *axial coding* to group items with relationships, such as opposition (*e.g.*, *defect finding* and *defect escaping -> defect finding*), subordination (*e.g.*, *defect finding* and *code quality assurance -> code quality assurance*) and causality (*e.g.*, *bias*, *conflict*, and *team awareness -> emotion*). In this step, we brought all the items (*benefits* and *challenges*) together.

In S_4, we performed *selective coding* to output core themes (quality attributes), *i.e.*, *quality assurance* and *emotion*.

In S_5, we validated and improved the two quality attributes with real-world review comments. The procedures are as follows.

Source 3: Real-World Examples. We analyzed review comments from (1) commercial projects from domestic and overseas teams at one global ICT enterprise (20K+, the experimental dataset was collected from them), and from five IT enterprises (1K+) and (2) OSS projects: Eclipse and Qt (3K+), to increase the validity, actionability, and generalizability of the quality attributes. Having analyzed a large number of real-world review comments, we found reviewers frequently express confusion [22] (*e.g.*, raising questions) in comments. In general, there are two major types of questions: (1) to understand motivations and implementations of code, *e.g.*, "*why printStackTrace is there?*"; and (2) to express uncertainty for defects, *e.g.*, "*is it a mandatory parameter?*". Although such review comments may not trigger code changes immediately, they help to (1) remind authors and other reviewers (if existed) of seeking further defects [63]; and (2) transfer project context, coding knowledge and experience between authors and reviewers [10]. In a nutshell, if there is any confusion in code review, it should be raised in time to prevent possible risks by the misunderstanding. By raising questions, reviewers can reduce misjudgment to some extent, or they would result in low-quality code as well as authors' negative feelings. Moreover, reviewers and authors help each other by asking

useful questions. Therefore, we use 'Question' (Que) to characterize such review comments.

Moreover, we found that review comments express two styles when working for 'quality assurance': (1) point out defects [34], *e.g.*, "*this field is not required*"; and (2) offer suggestions [6], *e.g.*, "*delete this line.. this is for debugging*". Therefore, we break 'quality assurance' (synthesized at the second stage) into 'Evaluation' (Eva) and 'Suggestion' (Sug), to describe the two styles of review comments, respectively. Review comments should be evaluated by associating four attributes together to avoid potential bias. For instance, if adopting 'evaluation' only, some reviewers may roughly indicate "*incorrect here*", but neither explain the reason behind, nor provide suggestions for correction. In this case, review comments offer very little help to code authors. Table 2 shows the ultimate quality attributes in our conceptual model. Note that the quality attributes (by answering descriptive questions) differ in a dichotomy (0–*No*, 1–*Yes*) and, for 'emotion', "negative" is annotated with 0, whilst "neutral" and "positive" are all annotated with 1.

RQ1 In Retrospect: By employing data triangulation, four quality attributes of review comments emerge: 'emotion', 'question', 'evaluation', and 'suggestion'. The quality grade of a review comment can be generated by associating the outputs of its four attributes (cf. Table 3).

4.1.2 Attribute Mappings. Given that binary evaluations (e.g., useful or not) often lead to disagreements, we establish four quality tiers for review comments in our conceptual model: 'excellent' (IV), 'good' (III), 'acceptable' (II), and 'poor' (I). Additionally, we manually annotate quality attributes instead of comment grades to train automated models, as it is easier to reach agreement on attributes than on grades, thus reducing inconsistency in annotation. Acknowledging that there can be 16 possible pairings between the four quality attributes and the four grades, and recognizing that each software organization may have unique criteria for evaluating review comments, we do not enforce strict mapping rules in our model. Rather we provide a sample mapping (see Table 3) from an industrial case study as a reference.

4.2 Automated Model

4.2.1 Technical Preliminary. As we are aiming to predict four attributes (labels) from a single review comment, we can consider this issue as a multi-label learning problem [82], also known as multi-label classification. Multi-label learning is a subfield of supervised learning. It differs from binary classification, where each example corresponds to one of two labels, and multi-class classification, where each example is associated with only one label from multiple, but mutually exclusive, classes. Multi-label learning allows an example to be linked with multiple labels. Each label represents a different classification task that can be correlated with others. This approach offers advantages such as enhancing data and computational efficiency and reducing the risk of overfitting [82].

4.2.2 Baseline Classifiers. We evaluated six popular text classifiers as potential candidates for the automated model in our EvaCRC system. These classifiers include Random Forests (RF) [8], TextCNN [40], TextRCNN [43], DPCNN [38], Transformer [69], and BERT [20]. Among these, RF is a traditional machine learning model based on pre-designed features, while the rest are deep learning models that simplify the process by avoiding laborious feature engineering tasks, such as feature extraction and selection [17, 44]. RF was included because it has proven effective in prior research [34, 57]. Though these models were primarily designed for single-label text classification, we adjusted their architecture (specifically their output units and loss functions) to suit our multi-label text classification needs. For the RF model, we first used Word2Vec [28] to convert a textual review comment into a vector, effectively replacing feature engineering, and then fed it into the RF model. During the training phase for each classifier, we finely tuned their hyperparameters, such as input length, to adapt them to our specific task.

4.2.3 Evaluation Metrics. In line with the EvaCRC system's approach of defining the main component of review comment evaluation as a multi-label text classification task, we used standard multi-label learning metrics [74] to assess classifiers: Hamming Loss (HL), Subset 0/1 Loss (01L), Macro-F1 (mF), and Macro-AUC (mA). Hamming Loss measures the proportion of labels that are incorrectly predicted, while Subset 0/1 Loss is stricter, requiring all labels for an example to be predicted correctly. Macro-F1 and Macro-AUC calculate the F1-score and AUC-score independently for each class, then average the results. This process ensures all classes are treated equally and helps evaluate how well classifiers handle class imbalance problems. Furthermore, as predicting a single label can be viewed as a binary classification problem, we used Precision (P), Recall (R), F1-score (F), and AUC (A) to evaluate the classifiers on this aspect. In terms of evaluation metrics, lower values of HL and 01L suggest better classifier performance, while higher values for other metrics indicate improved performance. To avoid bias, we conducted 5-fold cross-validation for each classifier.

5 STUDY II: EVALUATING EVACRC

This section reports on an industrial case study that empirically evaluates EvaCRC, including experiments and interviews.

5.1 Experimental Data

5.1.1 Data Collection. Given the significant variability in pull request comments (e.g., sentence length and element complexity) and the absence of such comments in many project's code files, we focused solely on inline review comments, as done in previous studies [6, 57]. As our aim is to create automated models with strong generalization capabilities, we did not restrict other project characteristics, such as the number of authors and reviewers.

5.1.2 Data Annotation. According to the configuration of our conceptual model, each review comment is assigned five labels. Four of these labels represent the quality attributes, which are manually annotated, while the fifth label indicates the quality grade, automatically determined based on predefined mappings. Table 3 illustrates the enterprise-specific mappings. The initial version of these mappings was drafted by three authors following discussions with practitioners from various software organizations. All the authors collaborated to create an intermediate version, and the final version was confirmed with professionals from the ICT enterprise.

Table 2: Descriptions of quality attributes

	Descriptive question	Rationale
Emo	Does it comment in a kind way?	Code review should evaluate code changes rather than authors.
Que	Does it ask questions or ask for confirmation?	Anything unclear in code review should be confirmed to prevent them from becoming risks. Also, knowledge transfer in code review should be two-way to benefit two sides.
Eva	Does it present evaluations or specify code weaknesses?	Code review should first detect and identify defects.
Sug	Does it provide suggestions for correction or prevention?	Code review is expected to help fix defects, improve quality, and improve developers' quality concerns.

Table 3: Exemplar mappings: quality attributes & grades

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14	M15	M16
Emo	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Que	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
Eva	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
Sug	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
	I	II	I	II	II	III	II	III	II	III	II	IV	IV	IV	II	IV

We manually annotated and carefully cross-checked 17,000 review comments (Section 6.1.1). Figure 2 shows the distributions of the experimental dataset, from which we observe that neither quality attributes nor grades are imbalanced.

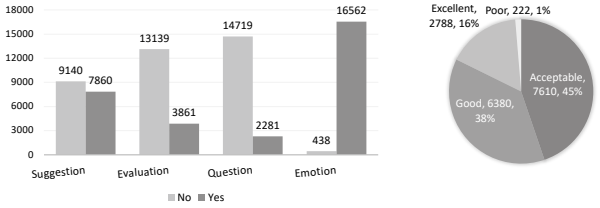

Figure 2: Distributions of the experimental dataset (left: attributes; right: grades)

5.2 Experimental Results and Analysis

5.2.1 Answer To RQ2.1 (Attribute's Perspective). Table 4 summarizes the performance of the six text classifiers in predicting quality attributes, with the best performances highlighted in bold. Our initial focus is on the precision and recall of each classifier. For the prediction of 'emotion', TextCNN stands out with the highest precision at 0.89, but its recall is relatively low at 0.44. DPCNN and Transformer have even lower recall rates, at 0.19 and 0.14 respectively, indicating a majority of the negative 'emotion' examples were inaccurately predicted. In our task, we consider the negative 'emotion' as the positive class, as our objective is to identify it. BERT, although it achieves a precision of 0.79, has a higher recall of 0.86, which places it first in terms of recall among the six classifiers. When predicting 'question', RF emerges as the best with the highest precision (0.96), closely followed by Transformer (0.95). BERT, with a precision of 0.94, ranks third but exhibits the highest recall (0.95). In terms of predicting 'evaluation', BERT's precision (0.77) is lower than that of Transformer (0.84), but BERT maintains second place. A similar pattern occurs in the prediction of 'suggestion': BERT is second in terms of precision (0.93), but it leads for recall (0.92).

Owing to its performance in recall, BERT significantly outperforms the other models in terms of F1-score. It registers scores of 0.82, 0.94, 0.75, and 0.92 for predicting 'emotion', 'question', 'evaluation', and 'suggestion', respectively. Additionally, because of its strong performance in maintaining a high true positive rate while minimizing false positives, BERT also outperforms the other text classifiers in terms of AUC. Furthermore, we found that BERT, along with the other text classifiers, generally performs better when predicting 'question' and 'suggestion' compared to the other attributes.

5.2.2 Answer To RQ2.2 (Grade's Perspective). Table 5 presents a summary of the performance of six text classifiers from the quality grade's perspective. We first focus on predicting 'excellent', BERT leads all the evaluation metrics (0.11 for Hamming loss, 0.37 for 0/1 loss, and 0.79 for macro-F1). Followed by RF whose performances are: 0.19 for Hamming loss, 0.65 for 0/1 loss, and 0.66 for macro-F1. Transformer performs worse than any others whose performances are: 0.31 for Hamming loss, 0.92 for 0/1 loss, and 0.52 for macro-F1. Due to the extremely high 0/1 loss, Transformer is unable to correctly predict 'excellent'. BERT continues to be a winner when predicting 'good', specifically, 0.06 for Hamming loss, 0.20 for 0/1 loss, and 0.63 for macro-F1. The suboptimal classifier is TextCNN with a Hamming loss at 0.09, a 0/1 loss at 0.33, and a macro-F1 at 0.55. The winner changes when predicting 'acceptable'. Specifically, Transformer that performs worst in predicting 'excellent' wins the lead over two evaluation metrics this time (0.02 for Hamming loss and 0.08 for 0/1 loss). The macro-F1 of Transformer is 0.42, just behind TextCNN (0.47) and BERT (0.57).

Finally, we focus on classifiers' performance in predicting 'poor', as it may suffer from most disagreements in practice. As shown in Figure 2, the numbers of 'excellent' and 'poor' review comments are significantly rare. That is, there is a serious class imbalance problem. Most classifiers do not work in this case. For instance, DPCNN's and Transformer's 0/1 loss is as high as 0.76 and 0.85, respectively. BERT is highly effective, however. BERT's Hamming loss and 0/1 loss are as low as 0.04 and 0.14, respectively, making it remarkably outperform other classifiers. The closest classifier is RF whose Hamming loss and 0/1 loss are 0.14 and 0.40, respectively. BERT also wins RF in terms of macro-F1 (0.48–0.44). To sum up, BERT in most cases is the most effective classifier from the perspective of either quality attributes or grades.

> **RQ2 In Retrospect:** The experiments with 17,000 real-world examples show that the BERT-based multi-label text classifier can effectively predict the quality attributes of review comments, significantly outperforming other classifiers on multiple metrics. We select it for the automated model.

5.3 Practitioners' Feedback

5.3.1 Methodology. We conducted focus group interviews [55] with practitioners to collect a variety of feedback of EvaCRC. The methodology of the interviews is as follows.

Step 1: Design&Preparation. Our protocol for the focus group interview consists of five aspects.

Objectives: We aim to evaluate and improve EvaCRC.

Questions: Three questions should be answered by interviewees.

- Q1: How useful of EvaCRC for improving code reviews? (ranging from 1 to 5 grades)
- Q2: What are the shortcomings of EvaCRC?
- Q3: How to address the shortcomings of EvaCRC?

Table 4: Performance summary of each text classifier from the attribute perspective

	RF				TextCNN				TextRCNN				DPCNN				Transformer				BERT			
	P	R	F	A	P	R	F	A	P	R	F	A	P	R	F	A	P	R	F	A	P	R	F	A
Emo	0.85	0.50	0.63	0.89	**0.89**	0.44	0.57	0.90	0.88	0.33	0.44	0.93	0.59	0.19	0.25	0.81	0.84	0.14	0.23	0.91	0.79	**0.86**	**0.82**	**0.98**
Que	**0.96**	0.80	0.88	0.98	0.94	0.86	0.89	0.98	0.92	0.86	0.89	0.98	0.78	0.71	0.74	0.93	0.95	0.83	0.88	0.98	0.94	**0.95**	**0.94**	**0.99**
Eva	0.65	0.40	0.49	0.86	0.76	0.47	0.58	0.88	0.74	0.50	0.59	0.89	0.53	0.41	0.46	0.76	**0.84**	0.21	0.33	0.87	0.77	**0.74**	**0.75**	**0.94**
Sug	0.83	0.87	0.85	0.93	0.91	0.80	0.85	0.94	0.91	0.80	0.85	0.95	0.85	0.73	0.78	0.89	**0.95**	0.65	0.77	0.94	0.93	**0.91**	**0.92**	**0.97**

Table 5: Performance summary of each text classifier from the grade perspective

	RF			TextCNN			TextRCNN			DPCNN			Transformer			BERT		
	HL	01L	mF*	HL	01L	mF	HL	01L	mF	HL	01L	mF	HL	01L	mF	HL	01L	mF
IV	0.19	0.65	0.66	0.19	0.64	0.63	0.20	0.64	0.60	0.26	0.76	0.48	0.31	0.92	0.52	**0.11**	**0.37**	**0.79**
III	0.10	0.34	0.54	0.09	0.33	0.56	0.09	0.33	0.55	0.13	0.44	0.44	0.11	0.41	0.51	**0.06**	**0.20**	**0.63**
II	0.06	0.17	0.41	0.03	0.11	0.47	0.04	0.11	0.42	0.06	0.18	0.29	**0.02**	**0.08**	0.42	0.03	0.11	**0.57**
I	0.14	0.40	0.44	0.13	0.43	0.43	0.16	0.54	0.40	0.27	0.76	0.32	0.24	0.85	0.30	**0.04**	**0.14**	**0.48**

* macro-AUC cannot be calculated if only true or false class examples exist.

Participants: Five interviewers and eight interviewees participated in the in-person focus group interview. The interviewers include one research professor, two doctoral researchers, and two master's students from the same academic institute. The interviewees include two executives (from the management division) and six representatives of the two projects studied (two developers and one leader from each). There were two steps for selecting the interviewees. First, we invited ten executives to participate in the interviews, two of which had the availability to meet us. Then the two executives invited further six interview participants because (1) they have more than 10 years of experience in development and code reviews; (2) they still participate in daily code reviews; (3) they are acknowledged as review experts by this enterprise; (4) and more importantly, they are in charge of two important projects whose review comments are selected for the live demonstration. With multiple roles of participants, we expect to receive relevant and insightful feedback in the interview.

Auxiliary Materials: Three types of documents were prepared for the interviews.

- D1: Technique Report. It elaborates on (1) background and related work; (2) evaluation designs (conceptual model and automated model, experimental settings, etc.); (3) evaluation results and analysis (automated model performance, error analysis, etc.); and (4) operation manuals (data preparation, automated model training and deployment, etc.).
- D2: Source Code Files. They are core code files for (1) data preprocessing (dataset shuffle and split, etc.); (2) BERT and other automated models' construction, training, and testing; (3) statistics and analysis (correlation analysis, etc.).
- D3: Evaluation Examples. They are textual review comments consisting of two parts: (1) experimental dataset (17,000 review comments); (2) two projects whose leaders should participate in the interviews (1,914 review comments). Each review comment is associated with four specific quality attributes and one overall quality grade.

Consents: Recording and disclosing the interview should be permitted without disclosing sensitive enterprise information or personal information, *e.g.*, names and evaluation results for exact projects.

Step 2: Execution. Two weeks before the interview, we delivered the technique report to the management division and project leader representatives at the ICT enterprise.

During the group interview, we first introduced the background of EvaCRC, the designs and implementations of EvaCRC, the designs and results of experiments, and the evaluation results of two internal projects that differ from the experimental projects. Then we ran EvaCRC for a live demonstration. The textual review comments (input), kernel code of EvaCRC (BERT-based text classifier), quality grade (output 1), and quality attributes (output 2) regarding each review comment were randomly checked by interviewees. Note that we displayed only the source code of EvaCRC to the interviewees. More precisely, the source code of the BERT-based multi-label text classifier which constitutes the automated model of EvaCRC. The two reasons are: (1) Evaluation: EvaCRC can be evaluated more rigorously if the source code is disclosed. As the technique report had been delivered before, the first key part of the interview is to evaluate the detailed implementation of EvaCRC (*i.e.*, source code). (2) Replication: Before the formal interview, the executives indicate that EvaCRC might be used at this enterprise if (a) EvaCRC performs well; and (b) the source code (*i.e.*, third-party libraries used) of EvaCRC are license-allowed. Therefore, we did not provide EvaCRC with a user interface but disclosed the source code only to make interviewees check third-party libraries and understand our model's architecture, dependencies, and input and output formats for replication.

Having finished the introduction and live demonstration, we first presented feedback to any comments or questions raised by the interviewees immediately and then asked the interviewees to answer the predefined three questions. Next, the executives started the discussion about how to apply EvaCRC from the aspects of data preparation, models' selection, training, deployment, output displays, etc., based on the existing platforms at the enterprise.

The interview lasts around two hours and a half. After that, we conducted several rounds of follow-up discussions with the executives and project representatives regarding improving and applying EvaCRC at this enterprise.

Step 3: Analysis. We received a wide range of feedback from participants based on their personal expertise, experience (story), and observations in the interview. Then we used the narrative analysis [12] method to qualitatively interpret interviewees' satisfaction, comments (concerns), and suggestions for improving and applying EvaCRC. At this stage, we performed inductive coding [65]. Finally, we concluded the interview by forming and checking the minutes.

5.3.2 Results. The introduction to EvaCRC was very well received. For the live demonstration, the interviewees were surprised by the ability to predict 'emotion' and were interested in mispredicted examples, even though the majority of outputs were consistent with their perception. For instance, "*Chances of NUll pointer!!!*", "*too*

many spelling mistakes. 'POSSITION'??" were correctly predicted to be with negative 'emotion'. "*Unnecessary casting to HTTP responseservlet*", "*Use method jQuery.isNumeric instead of two method call*" were correctly predicted to be 'evaluation' and 'suggestion', respectively. On the other hand, "*Indent error*" was incorrectly predicted to be 'suggestion', "*add the exception to the log. how to debug fi there is an error*" was incorrectly predicted to be 'question'. Then we discussed possible causes of errors (*e.g.*, low-quality dataset due to inconsistent standards for annotating review comments, without correction for model outputs) as well as correction and prevention strategies (*e.g.*, annotation checklists).

All interviewees recognized the value (*i.e.*, improving code review through evaluating code review comments) and effectiveness (*i.e.*, predicting quality attributes and grades of code review comments) of EvaCRC. Moreover, EvaCRC is designed to mitigate developers' notorious resistance to evaluations by providing explanations. Therefore, they all gave the highest usefulness rating. The main shortcomings of EvaCRC are (1) the inability to differentiate quality grades of very long review comments because they may be thoughtful and are graded to be 'excellent' almost; (2) the lack of confidence for the predicted quality attributes; (3) the lack of error (*i.e.*, any deviation from expectations)-correction mechanisms. The first shortcoming limits the usage scenario of EvaCRC, *i.e.*, inline review comments are ideal. For the second shortcoming, we could make the automated output not only the predicted labels but their inference probabilities. For the last, we could develop hand-crafted correction rules. Please refer to Section 6.2.2 (Ouput Correction&Improvement) for details.

Some interviewees raised questions about data annotation and validation, automated model selection and training, etc., and most importantly, the application of EvaCRC. We report on them in Section 6, which constitutes guidelines for applying EvaCRC.

> **RQ3 In Retrospect:** The practitioners' feedback received from an ICT enterprise indicates that software practitioners have recognized the value and effectiveness of EvaCRC. Moreover, they have provided insightful comments and suggestions for its improvement and application in practice.

6 DISCUSSION

In this section, we delve into lessons learned and suggestions for both researchers, who develop EvaCRC, and practitioners, who use EvaCRC. The lessons are primarily derived from our actions and observations as authors, while the suggestions are largely informed by feedback from practitioners—some of which have not been acted upon yet. We have organized this information primarily for clarity of presentation. It is important to note that the lessons learned could also serve as recommendations for future actions.

6.1 Lessons Learned

6.1.1 Annotating Review Comments. In our study, we have been able to gather a set of lessons that assist in annotation, such as identifying the presence of specific keywords, code elements, or analyzing the word count in a review comment. Additionally, there are existing tools (*e.g.*, [1, 79]) to assist annotation. However, these tools may not always perform as anticipated due to the inherent limitations and exceptions to hand-crafted rules [39].

For example, while we commonly interpret a '?' (question mark) as a clear indication of a 'question', it could also be part of a 'ternary if-else operator (b ? x : y)'. In such scenarios, the process becomes challenging as we are required to continually revise heuristic rules, modify regular expressions, and expand keyword dictionaries. Therefore, caution and flexibility are required when using automated tools and heuristics for annotation, taking into account the complexity and variability of code review comments.

Emotion. Review comments with positive words such as "*good, great, smart, cool, awesome*" likely indicate a positive 'emotion', while those containing negative words such as "*bad, poor, awful, terrible, shit, idiot, fool*" suggest negative 'emotion'. Additionally, the presence of multiple punctuation marks like "*..., ,,,, ???, !!!*" may hint at anger, satire, or criticism.

Question. When raising questions, review comments generally (1) contain interrogative keywords such as "*what, who, which, where, when, if, whether, how many, how long, how often*", etc.; and (2) consist of the general question or disjunctive questions; or, more simply, (3) contain the interrogative punctuation – "**?**". On the other side, review comments that ask for clarification or confirmation generally contain keywords such as "*check, examine, ensure, confirm, remember, make sure*".

Evaluation. When evaluating code changes, review comments generally (1) contain adjective keywords that indicate opinions or judgments, *e.g.*, "*redundant, useless, invalid, incorrect, insecure, obsolete, dangerous, illegal, unqualified, enough*". (2) contain adverbs of degree, *e.g.*, "*too, much, hardly, nearly, fairly, rather, almost, already*".

Suggestion. When offering suggestions, review comments generally (1) contain verb keywords that directly indicate what to do, or not to do, or how to do, *e.g.*, "*add, remove, delete, change, replace, format*", etc. (2) contain suggestive keywords, *e.g.*, "*suggest, recommend, how about, what about, why not, had better, should, avoid, forbid, use, no, don't, let's*".

Table 6: Examples of annotating code review comments

	C1*	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
Emo	1	1	1	0	1	1	1	1	1	1	1	1
Que	0	0	1	1	0	1	0	0	0	0	0	1
Eva	0	1	0	1	0	0	0	0	1	0	0	0
Sug	1	0	0	0	1	0	1	1	1	1	1	1
	III	III	II	II	III	II	III	III	IV	III	III	IV

* Please refer to Table 1 for complete review comments.

6.1.2 Ensuring Label Consistency. We have learned the following:

Prior Annotating. The benefits of prior annotating are (1) to warm up annotators as early as possible and, (2) to timely identify anomalies and violations, then adjust quality attributes as well as annotation checklists if needed, and further reach a consistent understanding among annotators.

Lock Checklists. Once reaching a consensus, it is recommended to temporarily lock annotation checklists in case of further extra debates. However, periodic revisits would be necessary since the dataset may have already shifted.

Pair Annotating. The functions of pair annotating are similar to pair programming or pair reviewing, which helps to stay focused and make timely corrections. Despite there being annotation

checklists, we strongly recommend pair annotating for handling borderline examples to ensure consistency.

Human-in-the-Loop. Human-in-the-loop [52] leverages machines and human knowledge to develop intelligent models. With human-in-the-loop learning, we first feed a few manually annotated examples into models for training, check and correct the predictive labels, and then feed into new examples. By repeating several iterations, it is likely to converge to consistent and correct labels.

Consistency Measure. Consistency measures the rate of agreement between multiple annotation outputs. We recommend the metrics such as Kappa coefficient to quantitatively measure the consistency rate. By adopting the above measures, we expect to construct the 'ground truth' dataset for training models.

6.1.3 Checking Example Consistency. *Example inconsistency* (a.k.a. dataset shift) occurs when the distribution of datasets differs between training and test set [54], making prediction models fail. It is different from label inconsistency as we assume that labels are completely correct and consistent in this case. The following are recommended techniques for detecting example inconsistency.

Descriptive Statistics. Descriptive statistics provide the basic information of examples (*e.g.*, comment words, affiliations) that summarize the dataset. Together with graphic analysis (*e.g.*, histogram and pie chart), they form the representations of datasets.

Hypothesis Testing. Statistical hypothesis testing works for assessing the dataset by using sample examples. For instance, we test whether the two groups of examples are equally distributed by leveraging Kolmogorov-Smirnov test [46] to detect dataset shift.

Error Analysis. Detecting example inconsistency by utilizing reliable models. Once the well-performed model does not work as expected, it is time to look into the examples' distribution as well as the labels' correction. We call it a model-based detection technique to distinguish the data-based detection techniques discussed above.

6.1.4 Analyzing Label Correlation. Through an analysis of the correlation between attribute labels, we can better understand reviewers' commenting habits, enabling us to improve EvaCRC. Figure 3 shows the correlation heatmap for 17,000 examples with a total of 68,000 binary class labels. The figure in each block represents the Kendall rank correlation coefficient τ of two variables (attribute labels). Note that all statistical hypothesis tests (null hypothesis: no correlation between two variables) were confirmed by checking the p-value (0.001) and therefore we do not present detailed p-values here. In general, if τ is less than 0.2, then two variables have few correlations; if τ is between 0.2 and 0.4, then two variables have a few correlations; otherwise, they have at least moderate correlations.

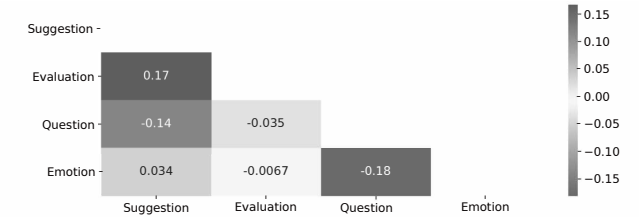

Figure 3: Correlation analysis of review comment labels

Figure 3 shows that, when serving suggestions, reviewers generally do not evaluate code changes or raise questions. Reviewers express positive emotions in most cases. In summary, all attribute labels have no strong correlations, which confirms the rationale behind EvaCRC– sharing common input representations but regarding each prediction task as independent. Due to these observations, we did not compare EvaCRC that is a "Algorithm Adaptation"-style approach (To adapt a single-label classification algorithm to the multi-label task by adjusting its cost function) to "Problem Transformation"-style approaches (To transform multi-label tasks to single-label tasks: single-class or multi-class), *e.g.*, Classifier Chain [58] (To regard each label as a part of a conditioned chain of single-class classification tasks) and Label Powerset [68] (To regard each label combination as a separate class with one multi-class classification task). Further discussion on "Algorithm Adaptation" and "Problem Transformation" are beyond the scope of this study.

6.2 Recommendation to Researchers

6.2.1 The Conceptual Model. The following are recommendations regarding two components of the conceptual model.

Attributes. Although there might be a number of attributes for evaluating review comments (*e.g.*, *change triggering*), they are at risk of disagreements from code reviewers. We have offered four attributes and experience in annotating. Since they have been validated with data triangulation, they are rigorous, credible, and can be directly transferred to other software organizations. We recommend tailoring attributes from the perspective of pragmatics rather than morphology and syntax (cf. Section 3.1) if necessary. Since our conceptual model consists of attributes and mappings, the more complex of attributes, the more complex mappings will be.

Mappings. We have established four-to-four mappings (four attributes and four grades). The case mappings (cf. Table 3) can be customized and tailored based on the software organizations' needs, *e.g.*, three quality grades. Again, we do not recommend the binary evaluation as it bears the risks of disagreements from developers.

6.2.2 The Automated Model. The following are recommendations regarding data, model, and correction.

Data Preparation. Due to insufficient review comments for several projects at this ICT enterprise, we have not separated projects for training and testing. Once a project accumulates enough comments, we suggest sampling an equal number from each for training and testing (*e.g.*, 400 for training and 100 for testing). It is also crucial that a project has diverse comments from multiple reviewers to ensure consistency across projects; without this, the automated model may have reduced accuracy and limited generalizability.

Model Selection&Training. We examined six leading text classifiers and found BERT to be exceptionally effective. Since BERT's debut, numerous variants like XLNet [78] and CodeBERT [27] have emerged. Exploring these BERT-based and newer models could enhance our automated model in the future. Additionally, training strategies like pretraining [32] and adversarial training [70] can further boost model accuracy and robustness.

Output Correction&Improvement. The evaluation largely depends on the automated model and set mappings. While models aid the process, they are not perfect, so correction strategies are essential. For example, comments with negative emotions can only

achieve 'acceptable', and without suggestions, they can not be 'excellent'. Even though we have not used linguistic characteristics in our model, they can enhance predictions; comments with over 20 words are deemed 'acceptable' at the minimum. Beyond rule-based corrections, examining mislabeled examples can highlight needed adjustments. Once rectified, these examples should be retrained in the model until accurately predicted. When it comes to improving the trustworthiness of outputs, we recommend outputting not only each quality attribute but also its confidence using predicted probability, *e.g.*, 'suggestion'-Yes-0.95.

6.3 Recommendations to Practitioners

For Organizations. Evaluations should be conducted in a way that minimizes pushback. For instance: (1) Display evaluation results in flexible formats, *e.g.*, online for quality attributes (reminder), offline for quality grades (evaluation); (2) Use relaxed formats to convey results, *e.g.*, emojis; (3) Offer clear explanations and best practices based on set mappings, *e.g.*, guiding reviewers on how to avoid a 'poor' grade or achieve an 'excellent' one; (4) Conduct comprehensive evaluations that go beyond just review comments, *e.g.*, review participation and coverage, and the number and severity of defects detected in review and in post-review activities, *e.g.*, testing; (5) Make code changes and review comments public rather than restricting access to specific projects or reviewers. In essence, evaluations should aim at enhancing future code and review quality while also increasing developers' awareness of quality issues. If not executed correctly, evaluations can be counterproductive.

For Developers. In addition to considering the recommendations (rationales) provided in Table 2, reviewers should prioritize producing high-quality code over only striving for high-quality review comments. While experienced developers might find it redundant to elaborate on minor code defects, such as naming conventions, newcomers benefit from detailed explanations and suggestions for rectification. Authors should ensure clarity in their code comments, change descriptions, reminders for areas of uncertainty, and self-review prior to submitting for a review. Additionally, it is important for authors to respond promptly to any questions or clarifications sought by reviewers. Only with these practices can the code review process be both effective and efficient.

7 THREATS TO VALIDITY

Construct Validity. The possible threat to construct validity may result from the measurement of review comment "quality". There is no standard measurement for this problem so far. To this end, we employed data triangulation to collect, synthesize, and validate quality attributes, aiming to develop a valid measurement of review comment "quality". These attributes have been empirically confirmed to be able to cover almost all review comments while providing clear differences (cf. Figure 2), and have been confirmed by the developers at a large ICT enterprise.

Internal Validity. In the second study, we strive to ensure that the experimental results (effect) should only be affected by the classifiers (cause). We identified three possible threats to internal validity. The first concerns example collection and pre-processing. We collected examples from the projects with multiple participants and long-term development periods, and formatted examples by removing project-aware marks (*e.g.*, defect and severity) to reduce their impacts on classifiers. The second is about example annotation. We have taken many measures to improve annotation reliability (cf. Section 6.1.1 and Section 6.1.2). The third relates to text classifiers' input representation. We used deep features that are automatically learned by models, rather than hand-crafted features that are manually engineered by researchers. Because we can hardly distinguish the cause-and-effect relationship between data and classifier performances if we mix two types of features.

External Validity. We have evaluated EvaCRC at a large ICT enterprise, which may be under threat to external validity. We have made efforts to mitigate possible impacts. When developing the conceptual model, we employed data triangulation to collect, synthesize, and validate quality attributes. They are expected to be language (*e.g.*, English, German, Chinese), project (*e.g.*, commercial, OSS), and organization-independent. The example mappings reflect the interests of the enterprise but could be adjusted. When performing experiments for developing the automated model, we selected review comments from a wide range of projects (*e.g.*, domestic and overseas, innersource and private). The BERT-based automated model could be updated to adapt to new data sources. Overall, the industrial case study shows the value and effectiveness of EvaCRC. More importantly, EvaCRC provides a paradigm for evaluating review comments in which both the conceptual model and automated models are tailorable. Such a paradigm is expected to be generalized to other contexts with little effort when referring to our experience and recommendations.

8 CONCLUSIONS

In our pursuit of creating an explainable, automated system for evaluating review comments, we actively gathered empirical evidence through data triangulation. This led us to formulate a conceptual model. Furthermore, we utilized multi-label learning to develop text classifiers, culminating in an automated model powered by BERT. A case study at a global ICT enterprise validated the efficacy of both our conceptual and automated models. Our work's significance extends beyond introducing the conceptual and automated EvaCRC models for code review evaluations. Crucially, we also offer software organizations a pragmatic, customizable approach and share tangible experiences related to ensuring code review quality through the evaluation of review comments.

DATA AVAILABILITY

The replication package (data for triangulation, experiments, and interviews) is available in https://doi.org/10.5281/zenodo.8297481.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (No.62072227, No.62202219), the National Key Research and Development Program of China (No.2019YFE0105500) jointly with the Research Council of Norway (No.309494), as well as the Key Research and Development Program of Jiangsu Province (No.BE2021002-2). Alberto Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project 200021_197227.

REFERENCES

[1] Toufique Ahmed, Amiangshu Bosu, Anindya Iqbal, and Shahram Rahimi. 2017. SentiCR: A customized sentiment analysis tool for code review interactions. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 106–111.

[2] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 712–721.

[3] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. 2016. Factors influencing code review processes in industry. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 85–96.

[4] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. 2016. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering* 21, 3 (2016), 932–959.

[5] Amiangshu Bosu, Jeffrey C Carver, Christian Bird, Jonathan Orbeck, and Christopher Chockley. 2016. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at Microsoft. *IEEE Transactions on Software Engineering* 43, 1 (2016), 56–75.

[6] Amiangshu Bosu, Michaela Greiler, and Christian Bird. 2015. Characteristics of useful code reviews: An empirical study at Microsoft. In *Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories*. IEEE, 146–156.

[7] Larissa Braz, Christian Aeberhard, Gül Çalikli, and Alberto Bacchelli. 2022. Less is more: Supporting developers in vulnerability detection during code review. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering*. ACM, 1317–1329.

[8] Leo Breiman. 2001. Random forests. *Machine Learning* 45, 1 (2001), 5–32.

[9] Nancy Carter. 2014. The use of triangulation in qualitative research. *Oncol Nurs Forum* 41, 5 (2014), 545–547.

[10] Maria Caulo, Bin Lin, Gabriele Bavota, Giuseppe Scanniello, and Michele Lanza. 2020. Knowledge transfer in modern code review. In *Proceedings of the 28th IEEE/ACM International Conference on Program Comprehension*. ACM, 230–240.

[11] Moataz Chouchen, Ali Ouni, Raula Gaikovina Kula, Dong Wang, Patanamon Thongtanunam, Mohamed Wiem Mkaouer, and Kenichi Matsumoto. 2021. Anti-patterns in modern code review: Symptoms and prevalence. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 531–535.

[12] F Michael Connelly and D Jean Clandinin. 1990. Stories of experience and narrative inquiry. *Educational Researcher* 19, 5 (1990), 2–14.

[13] John W Creswell and J David Creswell. 2017. *Research design: Qualitative, quantitative, and mixed methods approaches*. SAGE.

[14] Daniela S Cruzes and Tore Dybå. 2011. Recommended steps for thematic synthesis in software engineering. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement*. IEEE, 275–284.

[15] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. 2015. Code reviews do not find bugs. How the current code review best practice slows us down. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 27–28.

[16] Nicole Davila and Ingrid Nunes. 2021. A systematic literature review and taxonomy of modern code review. *Journal of Systems and Software* 177 (2021), 110951:1–30.

[17] Letian Deng and Yanru Zhao. 2023. Deep learning-based semantic feature extraction: A literature review and future directions. *ZTE Communications* 21, 2 (2023), 11–17.

[18] Norman K Denzin. 2012. Triangulation 2.0. *Journal of Mixed Methods Research* 6, 2 (2012), 80–88.

[19] Norman K Denzin. 2017. *The research act: A theoretical introduction to sociological methods*. Routledge.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. ACL, 29–35.

[21] Emre Doğan and Eray Tüzün. 2022. Towards a taxonomy of code review smells. *Information and Software Technology* 142 (2022), 106737:1–24.

[22] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2021. An exploratory study on confusion in code reviews. *Empirical Software Engineering* 26, 1 (2021), 1–48.

[23] Vasiliki Efstathiou and Diomidis Spinellis. 2018. Code review comments: Language matters. In *Proceedings of the 40th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, 69–72.

[24] Carolyn D Egelman, Emerson Murphy-Hill, Elizabeth Kammer, Margaret Morrow Hodges, Collin Green, Ciera Jaspan, and James Lin. 2020. Predicting developers' negative feelings about code review. In *Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering*. ACM, 174–185.

[25] Ikram El Asri, Noureddine Kerzazi, Gias Uddin, Foutse Khomh, and MA Janati Idrissi. 2019. An empirical study of sentiments in code reviews. *Information and Software Technology* 114 (2019), 37–54.

[26] Michael E Fagan. 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 15, 3 (1976), 182–211.

[27] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: 2020 Conference on Empirical Methods in Natural Language Processing*. ACL, 1536–1547.

[28] Gensim. 2023. Word2vec embeddings. https://radimrehurek.com/gensim/models/word2vec.html.

[29] GitLab. 2020. Code review guidelines. https://docs.gitlab.com/ee/development/code_review.html. (accessed 01.25.2023).

[30] Google. 2019. Google's engineering practices documentation. https://google.github.io/eng-practices/review/. (accessed 01.25.2023).

[31] Sanuri Gunawardena, Ewan Tempero, and Kelly Blincoe. 2023. Concerns identified in code review: A fine-grained, faceted classification. *Information and Software Technology* 153 (2023), 107054:1–14.

[32] Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A Smith. 2020. Don't stop pretraining: Adapt language models to domains and tasks. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. ACL, 8342–8360.

[33] DongGyun Han, Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, and Giovanni Rosa. 2020. Does code review really remove coding convention violations?. In *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 43–53.

[34] Masum Hasan, Anindya Iqbal, Mohammad Rafid Ul Islam, AJM Rahman, and Amiangshu Bosu. 2021. Using a balanced scorecard to identify opportunities to improve code review effectiveness: An industrial experience report. *Empirical Software Engineering* 26, 6 (2021), 129:1–34.

[35] Yang Hong, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, and Aldeida Aleti. 2022. CommentFinder: A simpler, faster, more accurate code review comments recommendation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 507–519.

[36] Yuan Huang, Xingjian Liang, Zhihao Chen, Nan Jia, Xiapu Luo, Xiangping Chen, Zibin Zheng, and Xiaocong Zhou. 2022. Reviewing rounds prediction for code patches. *Empirical Software Engineering* 27, 1 (2022), 1–40.

[37] Todd D Jick. 1979. Mixing qualitative and quantitative methods: Triangulation in action. *Administrative Science Quarterly* 24, 4 (1979), 602–611.

[38] Rie Johnson and Tong Zhang. 2017. Deep pyramid convolutional neural networks for text categorization. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*. ACL, 562–570.

[39] Robbert Jongeling, Proshanta Sarkar, Subhajit Datta, and Alexander Serebrenik. 2017. On negative results when using sentiment analysis tools for software engineering research. *Empirical Software Engineering* 22, 5 (2017), 2543–2584.

[40] Yoon Kim. 2014. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*. ACL, 1746–1751.

[41] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W Godfrey. 2015. Investigating code review quality: Do people and participation matter?. In *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution*. IEEE, 111–120.

[42] Andrey Krutauz, Tapajit Dey, Peter C Rigby, and Audris Mockus. 2020. Do code review measures explain the incidence of post-release defects? *Empirical Software Engineering* 25, 5 (2020), 3323–3356.

[43] Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. 2015. Recurrent convolutional neural networks for text classification. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence*. AAAI, 2267–2273.

[44] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.

[45] Zhixing Li, Yue Yu, Gang Yin, Tao Wang, Qiang Fan, and Huaimin Wang. 2017. Automatic classification of review comments in pull-based development model. In *Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering*. KSI, 572–577.

[46] Frank J Massey Jr. 1951. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association* 46, 253 (1951), 68–78.

[47] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering* 21, 5 (2016), 2146–2189.

[48] Nuthan Munaiah, Benjamin S Meyers, Cecilia O Alm, Andrew Meneely, Pradeep K Murukannaiah, Emily Prud'hommeaux, Josephine Wolff, and Yang Yu. 2017. Natural language insights from code reviews that missed a vulnerability. In *Proceedings of the 9th International Symposium on Engineering Secure Software and Systems*. Springer, 70–86.

[49] Marco Ortu, Bram Adams, Giuseppe Destefanis, Parastou Tourani, Michele Marchesi, and Roberto Tonelli. 2015. Are bullies more productive? Empirical study of affectiveness vs. issue fixing time. In *Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories*. IEEE, 303–313.

[50] Ali Ouni, Raula Gaikovina Kula, and Katsuro Inoue. 2016. Search-based peer reviewers recommendation in modern code review. In *Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution*. IEEE, 367–377.

[51] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. 2021. The impact of code review on architectural changes. *IEEE Transactions on Software Engineering* 47, 5 (2021), 1041–1059.
[52] Rahul Pandey, Hemant Purohit, Carlos Castillo, and Valerie L Shalin. 2022. Modeling and mitigating human annotation errors to design efficient stream processing systems with human-in-the-loop machine learning. *International Journal of Human-Computer Studies* (2022), 102772:1–12.
[53] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. 2018. Information needs in contemporary code review. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 1–27.
[54] Stephan Rabanser, Stephan Günnemann, and Zachary C Lipton. 2019. Failing loudly: An empirical study of methods for detecting dataset shift. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran, 1396–1408.
[55] Fatemeh Rabiee. 2004. Focus-group interview and data analysis. *Proceedings of the Nutrition Society* 63, 4 (2004), 655–660.
[56] Janani Raghunathan, Lifei Liu, and Huzefa H Kagdi. 2018. Feedback topics in modern code review: Automatic identification and impact on changes. In *Proceedings of the 30th International Conference on Software Engineering and Knowledge Engineering*. KSI, 598–597.
[57] Mohammad Masudur Rahman, Chanchal K Roy, and Raula G Kula. 2017. Predicting usefulness of code review comments using textual features and developer experience. In *Proceedings of the 14th IEEE/ACM International Conference on Mining Software Repositories*. IEEE, 215–226.
[58] Jesse Read, Bernhard Pfahringer, Geoff Holmes, and Eibe Frank. 2011. Classifier chains for multi-label classification. *Machine Learning* 85, 3 (2011), 333–359.
[59] Peter C Rigby and Christian Bird. 2013. Convergent contemporary software peer review practices. In *Proceedings of the 21st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 202–212.
[60] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: A case study at Google. In *Proceedings of the 40th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*. ACM, 181–190.
[61] Forrest Shull, Janice Singer, and Dag IK Sjøberg. 2007. *Guide to advanced empirical software engineering*. Springer.
[62] IEEE Computer Society. 2008. IEEE 1028-2008 - IEEE standard for software reviews and audits. https://standards.ieee.org/standard/1028-2008.html. (accessed 12.15.2022).
[63] Davide Spadini, Gül Çalikli, and Alberto Bacchelli. 2020. Primers or reminders? The effects of existing review comments on code review. In *Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering*. ACM, 1171–1182.
[64] Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. 2019. Test-driven code review: An empirical study. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering*. IEEE, 1061–1072.
[65] David R Thomas. 2006. A general inductive approach for analyzing qualitative evaluation data. *American Journal of Evaluation* 27, 2 (2006), 237–246.
[66] Veronica A Thurmond. 2001. The point of triangulation. *Journal of Nursing Scholarship* 33, 3 (2001), 253–258.
[67] Andrea C Tricco, Jesmin Antony, Wasifa Zarin, Lisa Strifler, Marco Ghassemi, John Ivory, Laure Perrier, Brian Hutton, David Moher, and Sharon E Straus. 2015. A scoping review of rapid review methods. *BMC Medicine* 13, 1 (2015), 1–15.
[68] Grigorios Tsoumakas and Ioannis Vlahavas. 2007. Random k-labelsets: An ensemble method for multilabel classification. In *Proceedings of 18th European Conference on Machine Learning*. Springer, 406–417.
[69] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st Conference on Neural Information Processing Systems*. Curran, 5998–6008.
[70] Dilin Wang, Chengyue Gong, and Qiang Liu. 2019. Improving neural language modeling via adversarial training. In *Proceedings of the 36th International Conference on Machine Learning*. PMLR, 6555–6565.
[71] Dong Wang, Yuki Ueda, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. 2021. Can we benchmark code review studies? A systematic mapping study of methodology, dataset, and metric. *Journal of Systems and Software* 180 (2021), 111009:1–18.
[72] Dandan Wang, Qing Wang, Junjie Wang, and Lin Shi. 2021. Accept or not? An empirical study on analyzing the factors that affect the outcomes of modern code review?. In *Proceedings of the 21st IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 946–955.
[73] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.
[74] Xizhu Wu and Zhihua Zhou. 2017. A unified view of multi-label performance measures. In *Proceedings of the 34th International Conference on Machine Learning*. PMLR, 3780–3788.
[75] Pavlína Wurzel Gonçalves, Gül Çalikli, and Alberto Bacchelli. 2022. Interpersonal conflicts during code review: Developers' experience and practices. *Proceedings of the ACM on Human-Computer Interaction* 6, CSCW1 (2022), 1–33.
[76] Lanxin Yang, Jinwei Xu, Yifan Zhang, He Zhang, and Alberto Bacchelli. 2023. Data and materials. https://doi.org/10.5281/zenodo.8297481.
[77] Lanxin Yang, He Zhang, Fuli Zhang, Xiaodong Zhang, and Guoping Rong. 2022. An industrial experience report on retro-inspection. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 43–52.
[78] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. XLNet: Generalized autoregressive pretraining for language understanding. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019*. Curran, 5754–5764.
[79] Qingshuang Yu, Jie Zhou, and Wenjuan Gong. 2019. A lightweight sentiment analysis method. *ZTE Communications* 17, 3 (2019), 2–8.
[80] Fiorella Zampetti, Saghan Mudbhari, Venera Arnaoudova, Massimiliano Di Penta, Sebastiano Panichella, and Giuliano Antoniol. 2022. Using code reviews to automatically configure static analysis tools. *Empirical Software Engineering* 27, 1 (2022), 1–30.
[81] Farida El Zanaty, Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. 2018. An empirical study of design discussions in code review. In *Proceedings of the 12th IEEE/ACM International Symposium on Empirical Software Engineering and Measurement*. ACM, 1–10.
[82] Minling Zhang and Zhihua Zhou. 2014. A review on multi-label learning algorithms. *IEEE Transactions on Knowledge and Data Engineering* 26, 8 (2014), 1819–1837.

Received 2023-02-02; accepted 2023-07-27