# Understanding Automated Code Review Process and Developer Experience in Industry

Hyungjin Kim
hjkim17.kim@samsung.com
Samsung Research
Seoul, Korea

Yonghwi Kwon
yhwi.kwon@samsung.com
Samsung Research
Seoul, Korea

Sangwoo Joh
sangwoo.joh@samsung.com
Samsung Research
Seoul, Korea

Hyukin Kwon
hyukin.kwon@samsung.com
Samsung Research
Seoul, Korea

Yeonhee Ryou
yeonhee.ryou@samsung.com
Samsung Research
Seoul, Korea

Taeksu Kim
taeksu.kim@samsung.com
Samsung Research
Seoul, Korea

## ABSTRACT

*Code Review Automation* can reduce human efforts during code review by automatically providing valuable information to reviewers. Nevertheless, it is a challenge to automate the process for large-scale companies, such as Samsung Electronics, due to their complexity: various development environments, frequent review requests, huge size of software, and diverse process among the teams.

In this work, we show how we automated the code review process for those intricate environments, and share some lessons learned during two years of operation. Our unified code review automation system, *Code Review Bot*, is designed to process review requests holistically regardless of such environments, and checks various quality-assurance items such as potential defects in the code, coding style, test coverage, and open source license violations.

Some key findings include: 1) about 60% of issues found by *Code Review Bot* were reviewed and fixed in advance of product releases, 2) more than 70% of developers gave positive feedback about the system, 3) developers rapidly and actively responded to reviews, and 4) the automation did not much affect the amount or the frequency of human code reviews compared to the internal policy to encourage code review activities. Our findings provide practical evidence that automating code review helps assure software quality.

## CCS CONCEPTS

• **General and reference** → **Empirical studies**; • **Software and its engineering** → *Application specific development environments*; *Collaboration in software development*; Software defect analysis; Empirical software validation.

## KEYWORDS

code review, review bot, code review automation, static analysis


*ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore*

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3558950

**ACM Reference Format:**
Hyungjin Kim, Yonghwi Kwon, Sangwoo Joh, Hyukin Kwon, Yeonhee Ryou, and Taeksu Kim. 2022. Understanding Automated Code Review Process and Developer Experience in Industry. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3540250.3558950

## 1 INTRODUCTION

*Code Review Automation* is an essential activity to reduce human effort in modern code review [22]. It generally contains a process of writing code reviews through a predefined set of tools. Many studies have been conducted on code review automation, or *bots*, and lots of open-source projects use automation tools to support code review. This automation allows developers to focus on understanding the intention of code changes, and spend more time discussing the architectural design or domain-specific problems.

In spite of its convenience, code review automation in real-world industry faces several challenges. *Samsung Electronics* manufactures many types of electronic devices such as mobile phones, network equipment, smart TVs, and home appliances. Therefore, many projects in Samsung are different in their development environments, domains, sizes, programming languages, and processes. The automation is more challenging in these settings.

The first challenge is the variety of development environments. For example, each development group uses different review system to meet its own business requirements, including *GitHub* [11], *Helix Swarm* [18], or *Gerrit* [20]. Each review system offers unique user experiences and functionalities through their APIs. For instance, developers can communicate with richer information in GitHub using Markdown and HTML format messages. Other systems, however, lack this feature.

Frequent review requests is the second challenge . This scalability issue is crucial to the industry's development process because an automated system with poor performance may become the bottleneck. Every day, in *Samsung Electronics*, approximately ten thousands of review requests are created from more than 10,000 projects, and about 1.4 million lines of code are added and about one million removed.

The last challenge has to do with the diversity of development processes among the teams. To ensure software quality, each team defines its own assurance policies and uses various analysis tools

for this purpose. For example, there are some teams that perform static analysis as part of Continuous Integration (CI), but other groups use the tools during the integration testing phase. Some groups are using Java and Kotlin as programming languages, but most are using C/C++. Therefore, different tools are required for different languages.

In order to overcome these challenges, we developed *Code Review Bot*, a unified code review automation system for Samsung Electronics . It supports various review systems that are used in the company. In addition, it provides a variety of services such as code change size measurements in review requests, typo checks, potential defect detection, and coding style checks.

We conducted a survey to collect feedback from developers, as well as a quantitative analysis how much the automation reduces their efforts by examining the number and length of review comments. Over 70% of developers are satisfied with our system, according to the survey. Most developers agreed that *Code Review Bot* is helpful for improving their code quality. Also, almost all respondents thought code review automation could reduce 30% of their efforts. More than 60% of defects were fixed, as our system motivated developers to focus on improving code quality.

Key contributions are:

- We present *Code Review Bot*, an automatic code review system. It provides a unified interface and a unique user experience in the code review activity.
- *Code Review Bot* is scalable, so can provide feedback for each review request quickly.
- *Code Review Bot* supports various types of analyses, including commit size measurements, typo checks, potential defect detection, coding style checks, reporting unit tests, test coverage measurements, and architecture metrics visualization.
- We evaluated our system qualitatively and quantitatively. We found that more than 70% of developers are satisfied with our system, and more than 60% defects have been fixed by developers.

# 2 CHALLENGES IN CODE REVIEW AUTOMATION AT SAMSUNG ELECTRONICS

Automating code review helps to improve productivity, but adapting automation to large-scale industries can be challenging. In this section, we discuss the challenges associated with automating the process, especially within Samsung Electronics.

## 2.1 Variety of Development Environments

Considering the complex environments in large organizations, it is essential for code review automation systems to be designed with great care. We use many programming languages, two different source code management systems, and several review systems at Samsung Electronics.

Various programming languages are used in the company. 39% of developers use C/C++, 31% use Java/Kotlin, 17% use Python, 7% use JavaScript (including TypeScript), and 4% use C# and so on. An automation system should be able to handle this variety.

We also use two kinds of source code management systems (SCMs): Git and Perforce Helix Core. Development teams choose SCMs based on factors such as project sizes, target platform, and build environments. Perforce Helix is a centralized tool with powerful branch-out functionalities and versioning features suitable for product lines in large-scale projects. On the other hand, Git is a distributed system that supports more agile development processes rather than Perforce Helix.

Moreover, we utilize several code review systems: GitHub, Gerrit, and Helix Swarm. Some teams use Perforce Helix as their SCM, so they should conduct the code review using Helix Swarm. Other teams use Git and GitHub for the SCM and the code review system, respectively. There are also a number of teams that use Gerrit as their review system to develop Tizen platform or Android apps, because they are working with public open-source developers who are familiar with that system.

## 2.2 Frequent Review Requests

In addition, the frequent request for code reviews makes it difficult to automate the process. Especially for large-scale industries, performance and rapid response are the key requirements of automation tools [16].

In Samsung Electronics, thousands of review requests are created in a day from more than ten thousand projects, and about 1.4 million lines of code are added, and one million lines are removed every day. The median number of lines added and removed is 35 and 8, respectively. The mean value for added lines is 1,373 with a standard deviation of 8.70 and that for removed lines is 757 with a standard deviation of 4.75.

## 2.3 Diversity of Processes

Due to 2.1 and 2.2, the development processes vary among teams. Different programming languages require different tools for static analysis. Pylint, for example, is used for analyzing Python code, and FindBugs is used for Java. Different review systems also involve differnt Continuous Integration (CI) tools. Tools used in Samsung Electronics and their characteristics are as follows.

- **Static analyzers** Static analysis tools try to find useful information such as potential defects using only source code. The tools are not only used by open-source projects, but also by many industries in practice to validate and verify the code base in spite of false positives [3, 13]. We also use open-source static analyzers (e.g., Cppcheck, Pylint, FindBugs, etc.) as well as private in-house ones.
- **Unit testing tools** Unit tests are used to ensure that the software behaves as intended. The tools for running unit tests and checking test coverage vary by programming languages. The way we show the test results for developers varies by review systems too.
- **Architecture metrics instrument** The use of software architecture metrics is generally considered as desirable for project management and evaluation [9]. We measure the metrics with private in-house tools. The metrics provide an overview of architectural attributes related to reliability and maintainability.
- **Open source license validator** Open source license violation becomes one of the most important issues. We also check compliance issues with private validation tools.

- **Coding style checker** Coding style checkers detect violations of developer-defined or pre-defined rules. These tools allow projects to keep consistent aesthetics. Some tools also suggest patches to comply with the rule. Like static tools, different programming languages require different style checkers, and different review systems require different kinds of feedback.

Different review systems also involve different Continuous Integration (CI) tools. For instance, a CI tool designed for GitHub may analyze source code in the head commit of the pull request to provide useful insights for reviewers. On the other hand, review process of Gerrit, using *Changes* and *Patch Sets* for change needs different process configuration of CI than GitHub.

## 3 CODE REVIEW BOT: AUTOMATIC CODE REVIEW SYSTEM

We developed a code review automation system for Samsung Electronics, *Code Review Bot* (*CRB*), which helps to create a unified code review process regardless of the tools and systems.

Design goals of *Code Review Bot* are as follows:

(1) *(Extensibility)* One of our major goals is to make *CRB* as easy as possible for quality tool developers to service their tools to different review systems. This is achieved by abstracting all the review systems used in the company as *Abstract Review System*, and providing unified APIs.
(2) *(Convenience) CRB* provides an accessible user interface for developers to configure the options of the services they use. We try to simplifying the complex configuration of all the tools in use with a few clicks. In addition, for all review systems, *CRB* gives its feedback in a consistent way.
(3) *(Scalability)* In order to handle the massive requests for review in Samsung Electronics , *CRB* should be scalable. We accomplish this by distributing the requests among several workers and processing each task asynchronously.

### 3.1 Overall Architecture

Figure 1 shows an overall architecture of *CRB*. When a developer creates a review request using a review system, the review system triggers *CRB* to start analyses. Receiving the request, each service of *CRB* inspects code changes from the review request and generates review comments including its analysis results. The developer confirms comments via the review system by herself and checks detailed information from *CRB*. With the help of *CRB*, reviewers are able to concentrate on understanding semantics of the code change and make review comments without worry of simple errors such as typos, potential defects, or coding convention violations.

The main feature of *CRB* is a connection between review systems and services, each of them is located on both sides of *CRB* backend. *Adapter* module takes a role of communication between a review system and *CRB*. During the information exchange, *Abstract Review System* component abstracts concrete review systems and encapsulates their detail. *Abstract Review System* provides common interfaces for every review system and lets *CRB* easily extend new kinds of systems.

Invocation of analysis services and scheduling between services are controlled by *Distributed Task Queue Worker* module. The worker controls an execution order and aggregates analysis results. Moreover, it is designed for parallel processing to boost up the performance of *CRB*. To meet the requirement, the worker component is implemented with *Celery* – a simple, flexible, and reliable distributed system for Python language [27]. Moreover, services need to be activated or turned off during an operation with a small amount of effort. Thus, we designed *Plugins* component based on a microservice architecture style [25] for scalability and extensibility.

Figure 2 illustrates the plugin architecture of *CRB*. Each plugin matches a relevant analysis service . Plugins are categorized based on the *invocation timing*, the *communication protocol* between a plugin and a relevant service, and the *execution type – synchronous or asynchronous*. Each category defines default behaviors such as an interpretation of a request, a service invocation, and a response generation.

Starting points of services are classified into three categories: when a review request is created ($S_1$), when other services have been finished ($S_2$), or when code fragments have been merged ($S_3$). Endpoints are separated into two groups: *synchronous analysis*, waiting for the finish of service since the analysis time is short enough ($E_1$), or *asynchronous analysis*, receiving an asynchronous result from a service via callbacks because of the long time-consuming analysis ($E_2$.) Based on the characteristics of services, we defined plugin classes: *RestSyncPlugin* ($S_1$ and $E_1$), *RestAsyncPlugin* ($S_1$ and $E_2$), *AfterAnalyzedPlugin* ($S_2$), and *AfterMergedPlugin* ($S_3$). As all *AfterAnalyzedPlugins* and *AfterMergedPlugins* are executed in synchronous, we did not define *AfterAnalyzedAsyncPlugin* and *AfterMergedAsyncPlugin*.

*Configuration Manager* module is also located inside the backend. The component manages settings and configurations of each project and service. The configurations are managed by users via *CRB Frontend*. *CRB Frontend* provides a user experience for developers to change preferences of their own projects. A developer can add a new project, select services to be applied, and set in detail options with a small amount of efforts.

### 3.2 Extensibility

We introduce an *Abstract Review System* to improve the extensibility of review systems. Each review system has its own APIs with which one can retrieve information about a general review request, code description, or code fragments. Moreover, writing a review comment or creation of code change suggestion can be also achieved via the APIs. An abstract review system, which is located inside CRB backend, is an abstraction of those APIs. It provides common interfaces and bridges interfaces with corresponding APIs of each review system. For instance, when the abstract review system receives a request to write a comment, regardless of review systems, *CRB* can write a comment to the review system by using an API of the abstract review system. *CRB* can easily address new type of review system with the help of the abstract review system.

Nevertheless, some variation points in review system still remain. For instance, rendering comments containing HTML or markdown may be unavailable for some review systems such as Gerrit. GitHub provides an API to retrieve the list of reviewers but Helix Swarm does not. It is important for the abstract review system to judge

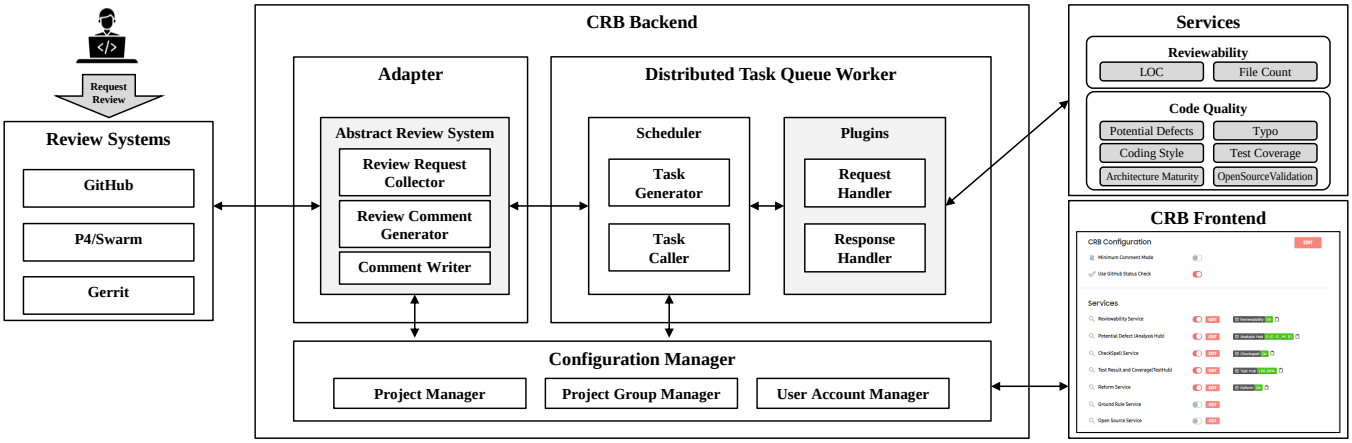



**Figure 1: Overall Architecture of *Code Review Bot***

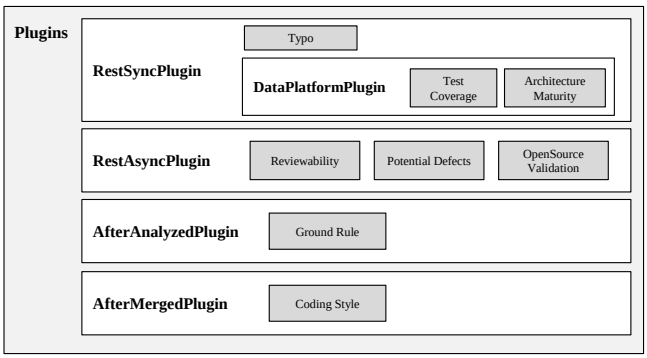



**Figure 2: Plugin Architecture**

whether the targeted review system supports certain functionality or not. Variations are controlled by an inheritance, and each instance filters out unavailable features by itself.

Since enough abstraction is applied, a service developer, someone who wants to create a new service does not have to consider the representation of each review system. She can concentrate on the implementation of a successful service only without any worry about the user interface compatibility. Moreover, if Samsung Electronics adapts a new review system, the abstract review system can be applied to *CRB* in convenient way.

## 3.3 Convenience

*CRB* provides several pre-defined analyses services. Developers can easily configure the options of the services with a few clicks.

In Addition, *CRB* returns each feedback in a consistent way to each review system. Depending on the sort of feedback, analyses services are as follows:

- **Inline comment services** Feedback of some services are delivered as inline comments. These services include **Potential Defects** service, **Typo** service, and **Open-Source Validation**. Defects detected by each tool are listed, and *CRB* indicates each line in which a defect has been found. Developers are able to identify problems and fix them more quickly with the aid of *CRB* . Project managers can easily turn on and off each service themselves and set configuration with little difficulty.
- **Summary comment services** The **Reviewability** service measures change metrics (e.g. the number of changed files or lines, etc.) and provides them to reviewers. **Test Coverage** service and **Architecture Maturity** service both run unit testing and code metrics measurements and provide information regarding overall code quality to developers. It is unfortunate that dynamic and static analysis for the measurement requires more effort from the project manager during the configuration. *CRB* lessens the human effort by providing 1) utility functionality for gathering test results and transforming them into a unified format for mainstream test frameworks and 2) pre-defined containers that can be used for running the metric measurements.
- **Code suggestion services** There are some services that suggest code changes to resolve certain issues. **Coding Style** service checks style violations, and sometimes creates a review request to fix those violations. The reason for this is that many coding style tools reconstruct the entire target file to comply their style, so inline comment feedback is not always sufficient for resolving violations. **Typo** service also creates a code fix suggestion and helps developers easily correct misspells.
- **CI-based services** Some services require execution environments to get their results, which means they need build systems or Continuous Integration systems. These services include **Potential Defects**, **Architecture Maturity**, and **Test Coverage** services. They uses Docker, Jenkins or GitHub Actions to build or test the submitted codes. Some divisions made their own build systems, so *CRB* triggers to start by using the systems. Or, other divisions use CI systems which are triggered by review systems directly such as GitHub Actions. At that time, *CRB* does not behave anything, but waits the results of CI systems.

## 3.4 Scalability

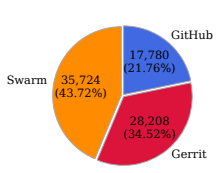
One of the most painful points in applying a review automation system to a large-scale industry is the performance problem by the lack of scalability. A large-scale project sometimes consumes huge amount of resources in analyses and execution time explosion may burden developers. We applied a microservice architecture style to *CRB* to overcome the limitation. The microservice architecture is known as a good solution for a scalable system [12].

Each services are deployed in physically separated nodes and communicate with relevant plugins via HTTP protocol. Parameters and environments are shared via *Redis* [17], a distributed in-memory database and a invocation scheduling is performed by *Celery* [27], a distributed task queue. Each service operation is processed independently in parallel.

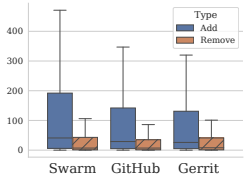
Deployment based on microservices achieves high level of scalability in a service expansion. Separation of services from the platform (i.e. *CRB* backend) makes the scaling out and up in convenient way and *CRB* successfully adapted into Samsung Electronics without performance problems.

## 3.5 Status of Usage

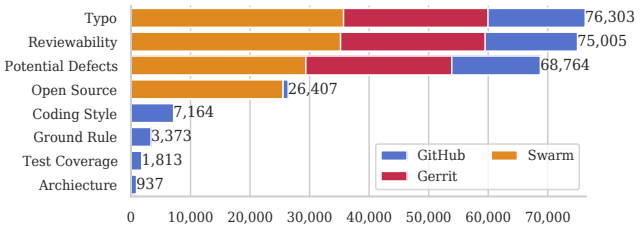
*CRB* has been applied in Samsung Electronics for about two years from 2019. More than eight thousand review requests from 6,439 projects have been created during the period and 2.7 million service executions happened. As the registration date of each project is different, to compare the portion of review systems and frequently executed services, we choose a single month (August 2021) and calculated the statistics.




(a) Review Requests




(b) Line Changes in Requests

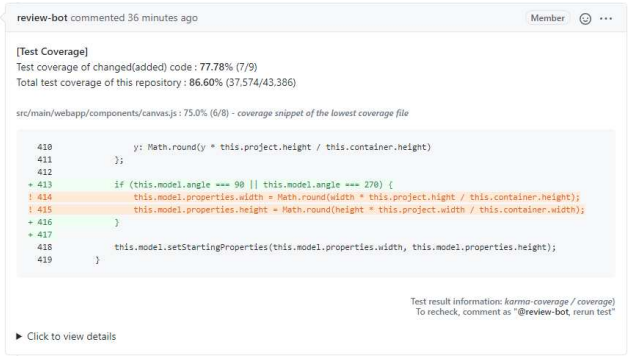



(c) Review Requests by Services

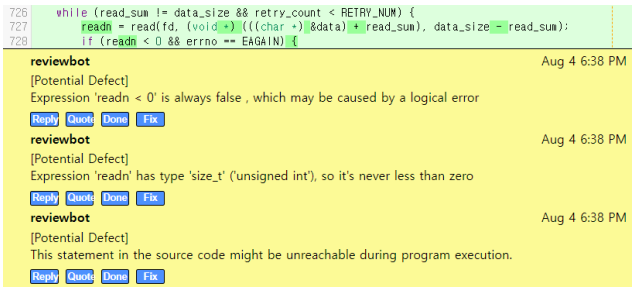
**Figure 3: Review Request Statistics in August 2021**

Figure 3a shows the number of review requests in a month. Totally 81,712 review requests were created and the average request count in a day is 3,714 (22 working days in a month). About 80% of requests were created from Helix Swarm and Gerrit. Figure 3b shows changed lines in a review request during the period. Median values of added lines per review request for Helix Swarm, GitHub, and Gerrit are 41, 29, and 26, respectively. Median values of removed lines are 7, 7, and 8, respectively.

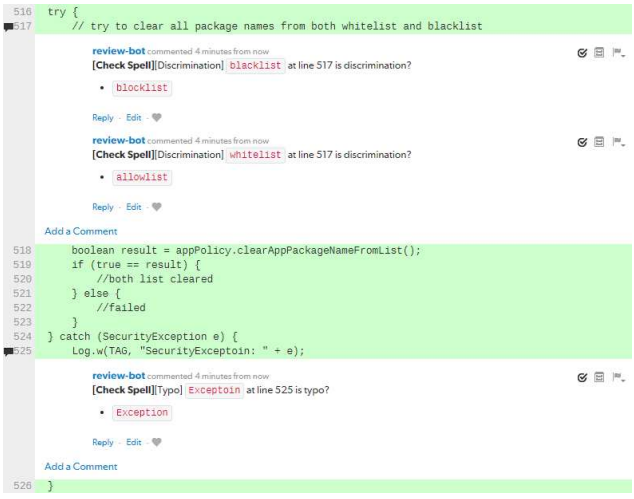
*Typo*, *Reviewability* and *Potential Defects* are the most frequently used top three services (86.7%). Number of service calls are illustrated in Figure 3c. Some services such as *Coding Style*, *Test Coverage*, and *Architecture Maturity* takes lower portion than top four services because the services can be applied to neither Helix Swarm nor Gerrit.




(a) Comment of Test Coverage Service from GitHub




(b) Comment of Potential Defect Service from Gerrit




(c) Comment of Typo Service from Helix Swarm

**Figure 4: Comments by *Code Review Bot***

Examples of comments generated by *CRB* for each review system are shown in Figure 4. Figure 4a is an example of *Test Coverage* service, which shows line coverage for both total lines and changed lines. Uncovered lines of the review request are emphasized to call a reviewer's attention. Figure 4b shows a comment generated into a Gerrit review system which contains potential defects in a review request. Example of a Helix Swarm comment contains several typos is shown in Figure 4c.

# 4 EMPIRICAL STUDIES

## 4.1 Research Questions

We conducted qualitative and quantitative analyses to see how useful *Code Review Bot* is for developers in Samsung Electronics. We made a survey to understand developers' satisfaction with the qualitative evaluation and obtained data from code review comments from developers and analysis results from *CRB* for the quantitative evaluation. From the results, we performed an empirical study to answer following research questions:

- **RQ1**: How useful is *CRB* for developers, and how efficiently do developers use *CRB*?
- **RQ2**: Does *CRB* attract developers? That is, how many defects and alarms from the system are approved and fixed by developers?
- **RQ3**: Does *CRB* quantitatively reduce the review effort? Is there significant difference in the amount of code review written by reviewers before and after the automation?

## 4.2 Experiment Setup

To answer RQ1, we conducted a survey from 93 developers in Samsung Electronics with questionnaires listed in Table 1. The questionnaires focus on understanding feedback about a satisfaction and usage patterns on *CRB*.

**Table 1: Questionnaires about Usage of *Code Review Bot***

| Question |
|---|
| $Q_1$: Are you satisfied with *Code Review Bot*? |
| $Q_2$: Does *Code Review Bot* effectively reduce time and effort in the code review activity? |
| $Q_3$: Do you think how much time and effort of the code review are reduced by *Code Review Bot*? |
| $Q_4$: Do you think *Code Review Bot* is helpful in the improvement of the code quality? |
| $Q_5$: Which benefits do you expect with *Code Review Bot*? |
| $Q_6$: Which services of *Code Review Bot* are you satisfied the most? |

We counted the number of fixed defects and alarms which are detected by *CRB* for the quantitative evaluation to RQ2. A *fix rate* of a review request is defined as a ratio of *the number of fixed alarms* over *the number of detected alarms* in the request. We collected fix rates from whole review requests which have one or more alarms (e.g. typos and potential defects) over the entire period after applying *CRB*.

Regarding efforts of reviewers, we collected all reviews written by reviewers for one year. By calculating the number of reviews and lengths of strings in each comment, we tried to obtain a finding on whether there are significant changes between before and after *CRB* has been adapted. Review requests were categorized into two groups. The first group contains review requests that have at least one review comment written by *CRB* and the second group does not. We measured the average number of review comments per week. Average string lengths of review comments from the same reviewers were also calculated. Median values were used for average to mitigate data biases by outliers. We performed *Paired sample t-test* between two groups to answer RQ3. To collect enough data, we targeted reviewers who have created code reviews in both groups for more than ten weeks.

## 4.3 Effectiveness of *Code Review Bot*

We conducted a survey from developers in Samsung Electronics to understand the satisfaction on *CRB* and to see the effects of the automation. Total 1,108 contributors were chosen from projects which are using *CRB* and questionnaires about satisfaction, effectiveness, and expectation of *CRB* listed in Table 1 were sent to them. From 93 respondents, responses have been collected, and the response rate is 8.39%.

**Table 2: Responses to Questionnaires**

$Q_1$. I'm satisfied with adapting *Code Review Bot* to my projects.

| Strongly agree | Agree | Neutral | Disagree | Strongly disagree |
|---|---|---|---|---|
| 36 (38.7%) | 34 (36.6%) | 9 (9.7%) | 0 (0.0%) | 1 (1.1%) |

$Q_2$. *Code Review Bot* helps to reduce **time and effort in code review**.

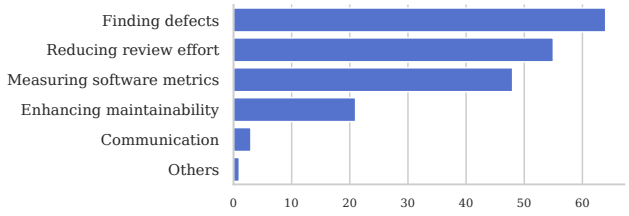
| Strongly agree | Agree | Neutral | Disagree | Strongly disagree |
|---|---|---|---|---|
| 38 (40.9%) | 28 (30.1%) | 10 (10.8%) | 3 (3.2%) | 0 (0.0%) |

$Q_4$. *Code Review Bot* helps to improve **code quality**.

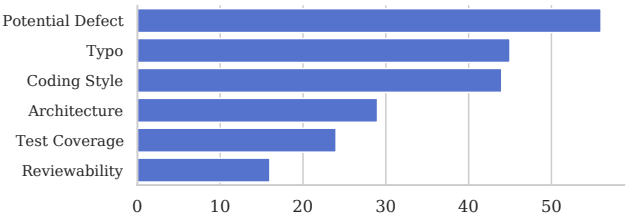
| Strongly agree | Agree | Neutral | Disagree | Strongly disagree |
|---|---|---|---|---|
| 37 (39.8%) | 37 (39.8%) | 4 (4.3%) | 1 (1.1%) | 0 (0.0%) |

Table 2 shows responses to questionnaires about the satisfaction with *CRB*. For a question about the overall satisfaction ($Q_1$), 75.3% of developers replied positive feedback. On the other hand, only 1.1% of developers answered negative opinions.

Next, 71.0% respondents agreed that *CRB* is helpful to reduce code review time and effort ($Q_2$) and only 3.2% disagreed. For the detailed question about the efficiency ($Q_3$), 36.6% reviewers thought that *CRB* reduces about 10% to 30% of their review time and 7.5% of them answered 30% to 50% of the review time might be reduced. 24.7% voted that less than 10% of the review effort could be reduced by *CRB*. In conclusion, more than 70% of respondents agree that *CRB* helps to save 30% or more efforts in code review. Next, $Q_4$ asked about the effect of *CRB* on the code quality, and 79.6% of respondents answered positive feedback.




**(a) Expected Impact of Code Review Automation**




**(b) Most Effective Services**

**Figure 5: Expectation and Services Usage**

From the result, we can make a conclusion that developers in Samsung Electronics are adapting *CRB* with a large amount of satisfaction. Especially, reviewers agree that *CRB* may efficiently reduce code review efforts and improves the code quality.

We collected additional responses to inspect developers' expectations to a code review automation and their usage pattern. Bacchelli and Bird showed that developers expect defect detection, code improvement, and finding better solutions from the code review [4] and Wessel et al. found that developers are also willing to get code review metrics and additional information such as analysis results (without having to go to another tool) followed by reducing maintenance effort from a code review automation tool [30].

Respondents expected several benefits from *CRB* ($Q_5$) as listed in Figure 5. We can see that similar results have been shown with prior studies, and *CRB* covers many parts of developers' expectations. Finally, developers choose the most useful services ($Q_6$) in the following order: *Potential Defects*, *Typo*, *Coding Style*, *Architecture Maturity*, and *Test Coverage*. Responses indicate that developers and reviewers are satisfied with code review automation in quality assurances and maintainability.

> **RQ1**. More than 70% developers are satisfying with *CRB* especially in the context of code quality improvement. Most of them feel that about 30% of code review effort may be reduced with the help of code review automation.

## 4.4 Motivating Quality Improvement by Automation

Regarding RQ2, we performed a quantitative analysis on code fixes related to defects and alarms found by *CRB*. By answering about the question, whether developers are motivated by automation or not, we may see the attraction effect of *CRB*.

Table 3 and 4 show fix rates of typos and potential defects found during a month, August 2021. Fix rate of typos from Swarm is 31.12% and the rate from Gerrit is 22.78%. On the other hand, fix rate from GitHub is only 5.51% – 793 requests over 14,384 requests contain typos detected by *CRB*.

**Table 3: Fix Rate of Typos**

| System | Fixed | Found | Fix rate | Found† | Fix rate† |
|---|---|---|---|---|---|
| GitHub | 793 | 14,384 | 5.51% | 1,092 | 72.62% |
| Swarm | 3,611 | 11,604 | 31.12% | 6,324 | 57.10% |
| Gerrit | 556 | 2,441 | 22.78% | 937 | 59.34% |
| Total | 4,960 | 28,429 | 17.45% | 8,853 | 59.38% |

Found† = Adjusted detected typo counts, Fix rate† = Adjusted fix rate

We found that developers tend to skip the re-execution in the case of typo. Developers do not feel a necessity to perform typo checking again because fixing the alarms are easy, clear, and there will be little possibility of introducing side effects. However, since potential defects detected by a certain static analysis tool, developers are used to explicitly run *CRB* again to confirm the result from the tool.

To complement the difference among review systems and services, we filtered out analysis results when *CRB* were executed just once in a review request. Found† and Fix rate† of Table 3 indicate the adjusted total number of typo and the adjusted fix rate, respectively. Adjusted fix rate of GitHub is 72.62% but adjusted fix rate of the other review systems are lower than GitHub. For all the review systems, we found that at least half of typo-detected requests were fixed.

**Table 4: Fix Rate of Potential Defects**

| System | Fixed | Found | Fix rate | Found† | Fix rate† |
|---|---|---|---|---|---|
| GitHub | 5,571 | 11,375 | 48.98% | 5,806 | 95.95% |
| Swarm | 3,745 | 11,059 | 33.86% | 6,094 | 61.45% |
| Gerrit | 1,067 | 2,026 | 52.67% | 1,394 | 76.54% |
| Total | 10,383 | 24,460 | 42.45% | 13,294 | 78.10% |

Found† = Adjusted detected typo counts, Fix rate† = Adjusted fix rate

In the case of potential defects, the adjusted fix rate of GitHub, Swarm, and Gerrit are 95.95%, 61.45%, and 76.54%, respectively. The aspect is similar with the case of typos but adjusted fix rates are higher than values of typos. The adjusted fix rate of GitHub shows very high value of 95.95%. That is, developers always trigger *CRB* again after they fix the potential defects.

> **RQ2**. Developers are strongly motivated to improve code quality from *CRB*. More than 60% defects and warnings are immediately fixed by developers.

## 4.5 Encouraging Reviewers

To investigate RQ3, we collected two kinds of data – 1) average counts of code review comments by the weekly and 2) average length of comments – during a year from September 2020 to August 2021. We performed a statistical hypothesis testing by using the data as a sample and estimate the fluctuations of the mean of the values. We set up a null hypothesis ($H_0$) and an alternative hypothesis ($H_1$) as follows:

- $H_0$: After adapting *CRB*, there is no difference in the code review size.
- $H_1$: After adapting *CRB*, there is a significant difference in the code review size.

A *paired sample t-test* with 95% confidence level was used for the hypothesis testing.

**Table 5: Differences in Code Review after Applying *CRB***

**(a) Review Size Metrics**

| System | By | $N$ | $\mu_0$ | $\sigma_0$ | $\mu_1$ | $\sigma_1$ |
|---|---|---|---|---|---|---|
| GitHub | C | 1,952 | 14.94 | 208.66 | 12.90 | 60.05 |
| Swarm | C | 815 | 13.62 | 119.19 | 9.36 | 64.79 |
| Gerrit | C | 625 | 107.00 | 1,900.80 | 26.70 | 234.90 |
| GitHub | L | 1,952 | 19.20 | 61.08 | 21.83 | 76.07 |
| Swarm | L | 815 | 109.86 | 159.71 | 100.44 | 149.99 |
| Gerrit | L | 625 | 31.95 | 39.33 | 31.81 | 26.11 |

C = By number of review comments
L = By string length of review comments
$\mu_0$, $\sigma_0$ = Before using *CRB*, $\mu_1$, $\sigma_1$ = After using *CRB*

**(b) Paired Differences**

| System | By | PDM | PDSD | 95% CI | $t$ | $p$ |
|---|---|---|---|---|---|---|
| GitHub | C | −2.04 | 198.56 | (−10.85, 6.78) | −0.45 | 0.651 |
| Swarm | C | −4.26 | 97.12 | (−10.93, 2.42) | −1.25 | 0.211 |
| Gerrit | C | −80.40 | 1,832.60 | (−224.30, 63.60) | −1.10 | 0.273 |
| GitHub | L | 2.63 | 62.06 | (−0.12, 5.39) | 1.87 | 0.061 |
| Swarm | L | −9.42 | 86.89 | (−15.39, −3.44) | $-3.09^{**}$ | 0.002 |
| Gerrit | L | −0.14 | 17.24 | (−1.50, 1.21) | −0.21 | 0.836 |

C = By number of review comments, L = By string length of review comments
PDM, PDSD = Mean and Standard deviation of paired differences
$^{*}p < 0.05$, $^{**}p < 0.01$

Table 5 shows the *paired sample t-test* result of the difference in means between before and after adopting *CRB*, including medians of average comment counts by weekly and medians of comment lengths by reviewers. Only one hypothesis $H_0$ can be rejected for all review systems – Swarm & Length case. In other words, except the one case, there are no significant differences whether *CRB* is applied or not. Table 6 is a detailed result for review system servers – two GitHub servers, four Swarm servers, and two Gerrit servers. Interestingly, about half of all cases reject $H_0$. All rejected L cases have a positive t-value. That is, adapting *CRB* affects reviewers to write *longer* comments.

Actually, one of the main reasons why average counts of review comments changed is infected by policies – what kinds of activities should be performed during the code review phase – of each development division. Regardless of adapting *CRB*, reviewers may review code more and more when any activities are conducted for them.

**Table 6: Differences in Code Review after Applying *CRB* for Each Review System**

| Server | System | By | $N$ | PDM | PDSD | $t$ | $p$ |
|---|---|---|---|---|---|---|---|
| $GH_1$ | GitHub | C | 1,863 | 4.50 | 38.73 | $5.02^{**}$ | 0.000 |
| $GH_2$ | GitHub | C | 89 | −138.90 | 906.90 | −1.44 | 0.152 |
| $SW_1$ | Swarm | C | 498 | −7.59 | 117.14 | −1.44 | 0.151 |
| $SW_2$ | Swarm | C | 136 | 9.45 | 34.13 | $3.23^{**}$ | 0.002 |
| $SW_3$ | Swarm | C | 155 | −5.67 | 62.35 | −1.13 | 0.259 |
| $SW_4$ | Swarm | C | 26 | −3.54 | 6.09 | $-2.96^{**}$ | 0.007 |
| $GR_1$ | Gerrit | C | 561 | −87.70 | 1,934.30 | −1.07 | 0.283 |
| $GR_2$ | Gerrit | C | 64 | −15.87 | 48.03 | $-2.64^{*}$ | 0.010 |
| $GH_1$ | GitHub | L | 1,863 | 2.54 | 63.44 | 1.73 | 0.084 |
| $GH_2$ | GitHub | L | 89 | 4.62 | 15.11 | $2.88^{**}$ | 0.005 |
| $SW_1$ | Swarm | L | 498 | −16.12 | 102.28 | $-3.52^{**}$ | 0.000 |
| $SW_2$ | Swarm | L | 136 | 5.21 | 14.27 | $4.25^{**}$ | 0.000 |
| $SW_3$ | Swarm | L | 155 | −0.13 | 16.22 | −0.10 | 0.923 |
| $SW_4$ | Swarm | L | 26 | −12.90 | 180.40 | −0.36 | 0.719 |
| $GR_1$ | Gerrit | L | 561 | −0.53 | 17.82 | −0.71 | 0.480 |
| $GR_2$ | Gerrit | L | 64 | 3.26 | 10.37 | $2.51^{*}$ | 0.015 |

C = By number of review comments, L = By string length of review comments
PDM, PDSD = Mean and Standard deviation of paired differences
$^{*}p < 0.05$, $^{**}p < 0.01$

**RQ3**. We can not say that *CRB* reduces code review effort dramatically. Some development division shows significant differences in the code review size but the others do not. Reviewers usually tend to affect by the policies and development release cycle of their development division more than the automation.

## 5 DISCUSSION AND THREATS TO VALIDITY

Although *CRB* provides unified interfaces for numerous review systems, variances in rendering review comments by each review system are still remain. Reviewers might be more attracted by a pretty written review comment with Markdown or HTML markups. Reactiveness from GitHub may be higher than other review systems because they only allow a plain text or a simple markup. It would be an interesting research topic on what kinds of differences among review systems arise because of the presentation divergence.

There is an internal validity in measuring the quality improvement motivation by *CRB* at section 4.4. In this study, we compared the change of detected defect counts after applying *CRB* to show how effectively *CRB* attracted developers, but developers of some projects already use several kinds of static analysis tools themselves. Thus, the evaluation results may not reflect the direct effect of *CRB* in the view of quality assurance. Different environments and build systems of each projects make obstacles to investigate how many developers start to use code analysis tools newly, not *CRB*. Nevertheless, we still believe that *CRB* successfully attracts quality improvement of source code because the convenience of *CRB* obviously aids developers to apply analysis tools for their projects no matter whether they have already run some tools themselves or not.

A lot of factors such as functionalities of a certain review system, development process, policies of a team, release plans or human resource organization might influence on the quality of source code

or code review manners. The fact causes a threat to the internal validity. In the study case, we gathered data with a consideration on not only review systems but also development teams and business divisions. Respondents are selected from various development teams from miscellaneous software domains to overcome the threat. Nevertheless, it is not able to guarantee that all research results in section 4 are not biased because of the factors.

The architecture of *CRB* in subsection 3.1 is generic and abstracted enough to be extended externally. Despite the extensibility of the architecture, it is not easy to directly apply *CRB* to open source projects or other industries rather than Samsung Electronics. Available services and schedulers need to be selected and built based on the consideration of the organization's culture, policy, and circumstance. Besides, as feedback from users also may be vary because needs of each develop team. The case study from Samsung Electronics may differ to the open source communities or other industrial cases.

## 6 RELATED WORK

Static analysis is an efficient method to find out defects, and applying static analyzers in review automation is effective to improve productivity. Singh et al. showed that about 16% of code review comments can be covered by static analysis tool [26]. From the case of Google, Sadowski et al. proved that providing static analysis results to developers in the code review phase is better than the test phase because developers are not willing to modify matured code after testing and release [21]. Facebook also applies a static analysis *Infer* in the review process [8]. Developers on Facebook were opened to fix the alarms and 70% of them were solved [10]. Static analyses are essential for code review automation, and *Code Review Bot* has been founded on static analyses.

Code review automation, named a *Bot*, is also widely adopted in open source communities [29]. Balachandran suggested *ReviewBot*, a review automation system based on static analyses, and successfully showed the feasibility of adapting to the real world [5, 6]. In this study, we developed *Code Review Bot* that provides several services based on the static analysis, testing, and fix suggestions and found its usefulness in the industrial area.

Google implemented *Tricoder*, a novel source code analysis framework [23] and melted it into their workflow [23]. Microsoft also combines several quality assurance tools into a framework and performs analysis with *CodeFlow* [7]. *Rosie* [19] is a fully automated review assistant that traverses whole repositories by itself and cleans up the codebase and fixes defects automatically in Google. *Code Review Bot* takes a role of quality assurance automation in Samsung Electronics based on developers' review requests.

Defect prediction is one of the most important next research topics of *Code Review Bot*. *JITBot* [15] successfully showed the feasibility of just-in-time defect prediction [14] in practice by adapting into GitHub. One of the most interesting and impressive contributions of *JITBot* is providing reasons to users with the help of explainable artificial intelligence [1]. Explanation and description of a prediction to end-users are important for tools aiming at real industry areas. For this reason, the study gave us an inspiration not only in the preparation of future work but also in the design of the user experience of *Code Review Bot*.

Several studies on review automation using machine learning have been performed. Asthana et al. suggested a novel reviewer recommendation automation based on a balance between commit and review history [2]. Saini and Britto showed a case in Ericsson on the application of review request prioritization based on a machine learning [24]. Tufano et al. experimented with deep learning techniques in the context of automating reviewer recommendations and program synthesis to assist reviewers [28]. State-of-the-art techniques indicate a direction of the next step of *Code Review Bot*.

## 7 CONCLUSION

We introduced a unified code review automation system, *Code Review Bot*. We have achieved extensibility and scalability for intricate, various, and diverse development environments in Samsung Electronics by using our abstract review system. We have been operating the system over two years for more than 10,000 projects, and shared some lessons learned.

We investigated both quantitative and qualitative analyses on satisfaction, usage, and effectiveness. Most of the developers were satisfied with *Code Review Bot*. Quality assurance services were the most helpful one, and developers rapidly and actively responded to more than 60% of reviews.

We believe that the code review automation helps not only for the quality of source code, but also for the rapid and worthwhile code review activities of reviewers. We plan to study on the correlation among services, review systems, and users reactions. We hope to discover some insights for more effective code review in the future.

## ACKNOWLEDGMENT

We would like to thank the anonymous reviewers, Yoonki Song, Geunsik Lim, JaeHyun Yoo, Youil Kim, and Joonbae Park, for their insightful comments and feedback that helped improve the paper. This work was supported in part by Intelligent Dev. Assistant & Quality Tool Development (RAJ0122ZZ-35RF), Samsung Research, Samsung Electronics Co., Ltd.

## REFERENCES

[1] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador García, Sergio Gil-López, Daniel Molina, Richard Benjamins, Raja Chatila, and Francisco Herrera. 2019. Explainable Artificial Intelligence (XAI): Concepts, Taxonomies, Opportunities and Challenges toward Responsible AI. arXiv:1910.10045 [cs.AI]

[2] Sumit Asthana, Rahul Kumar, Ranjita Bhagwan, Christian Bird, Chetan Bansal, Chandra Maddila, Sonu Mehta, and B. Ashok. 2019. WhoDo: Automating Reviewer Suggestions at Scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 937–945. https://doi.org/10.1145/3338906.3340449

[3] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. 2007. Evaluating Static Analysis Defect Warnings on Production Software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (San Diego, California, USA) *(PASTE '07)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/1251535.1251536

[4] Alberto Bacchelli and Christian Bird. 2013. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, NY, USA, 712–721.

[5] Vipin Balachandran. 2013. Fix-it: An extensible code auto-fix component in Review Bot. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, NY, USA, 167–172. https://doi.org/10.1109/SCAM.2013.6648198

[6] Vipin Balachandran. 2013. Reducing Human Effort and Improving Quality in Peer Code Reviews Using Automatic Static Analysis and Reviewer Recommendation. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, NY, USA, 931–940. https://doi.org/10.1109/ICSE.2013.6606642

[7] Christian Bird, Trevor Carnahan, and Michaela Greiler. 2015. Lessons Learned from Building and Deploying a Code Review Analytics Platform. In *Proceedings of the 12th Working Conference on Mining Software Repositories* (Florence, Italy) *(MSR '15)*. IEEE Press, NY, USA, 191–201. https://doi.org/10.1109/MSR.2015.25

[8] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods*, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi (Eds.). Springer International Publishing, Cham, 3–11.

[9] Shyam R. Chidamber and Chris F. Kemerer. 1991. Towards a Metrics Suite for Object Oriented Design. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications* (Phoenix, Arizona, USA) *(OOPSLA '91)*. Association for Computing Machinery, New York, NY, USA, 197–211. https://doi.org/10.1145/117954.117970

[10] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. https://doi.org/10.1145/3338112

[11] GitHub. 2021. GitHub. Retrieved from https://github.com/.

[12] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov. 2018. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software* 35, 3 (2018), 24–35. https://doi.org/10.1109/MS.2018.2141039

[13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, NY, USA, 672–681.

[14] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2013), 757–773. https://doi.org/10.1109/TSE.2012.70

[15] Chaiyakarn Khanan, Worawit Luewichana, Krissakorn Pruktharathikoon, Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, and Thanwadee Sunetnanta. 2020. *JITBot: An Explainable Just-in-Time Defect Prediction Bot.* Association for Computing Machinery, New York, NY, USA, 1336–1339. https://doi.org/10.1145/3324884.3415295

[16] Geunsik Lim, MyungJoo Ham, Jijoong Moon, and Wook Song. 2021. LightSys: Lightweight and Efficient CI System for Improving Integration Speed of Software. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE Press, NY, USA, 1–10. https://doi.org/10.1109/ICSE-SEIP52600.2021.00009

[17] Redis Ltd. 2021. Redis. Retrieved from https://redis.io/.

[18] Perforce. 2021. Helix Swarm: Free Code Review Tool For Helix Core. Retrieved from https://www.perforce.com/products/helix-swarm/.

[19] Rachel Potvin and Josh Levenberg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* 59, 7 (June 2016), 78–87. https://doi.org/10.1145/2854146

[20] Gerrit Code Review. 2021. Gerrit Code Review. Retrieved from https://www.gerritcodereview.com/.

[21] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at google. *Commun. ACM* 61, 4 (2018), 58–66.

[22] 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 International Conference on Software Engineering: Software Engineering in Practice* (Gothenburg, Sweden) *(ICSE-SEIP '18)*. Association for Computing Machinery, New York, NY, USA, 181–190. https://doi.org/10.1145/3183519.3183525

[23] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, NY, USA, 598–608.

[24] Nishrith Saini and Ricardo Britto. 2021. Using Machine Intelligence to Prioritise Code Review Requests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE Press, NY, USA, 11–20. https://doi.org/10.1109/ICSE-SEIP52600.2021.00010

[25] Mary Shaw and David Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, Inc., USA.

[26] Devarshi Singh, Varun Ramachandra Sekar, Kathryn T. Stolee, and Brittany Johnson. 2017. Evaluating how static analysis tools can reduce code review effort. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, NY, USA, 101–105. https://doi.org/10.1109/VLHCC.2017.8103456

[27] Ask Solem. 2018. Celery - Distributed Task Queue. Retrieved from https://docs.celeryproject.org/.

[28] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. 2021. Towards Automating Code Review Activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE Press, NY, USA, 163–174. https://doi.org/10.1109/ICSE43902.2021.00027

[29] Mairieli Wessel, Bruno Mendes de Souza, Igor Steinmacher, Igor S. Wiese, Ivanilton Polato, Ana Paula Chaves, and Marco A. Gerosa. 2018. The Power of Bots: Characterizing and Understanding Bots in OSS Projects. *Proc. ACM Hum.-Comput. Interact.* 2, CSCW, Article 182 (Nov. 2018), 19 pages. https://doi.org/10.1145/3274451

[30] Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A. Gerosa. 2020. What to Expect from Code Review Bots on GitHub? A Survey with OSS Maintainers. In *Proceedings of the 34th Brazilian Symposium on Software Engineering* (Natal, Brazil) *(SBES '20)*. Association for Computing Machinery, New York, NY, USA, 457–462. https://doi.org/10.1145/3422392.3422459