MASTER'S THESIS 2021

The Costs and Benefits of Acting on Program Analysis Results

Mattias Leifsson, Michael Pater

ISSN 1650-2884
LU-CS-EX: 2021-20

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2021-20

The Costs and Benefits of Acting on Program Analysis Results

Kostnader och fördelar med att agera på resultat från programanalys

Mattias Leifsson, Michael Pater

The Costs and Benefits of Acting on Program Analysis Results

Mattias Leifsson
ma5565le-s@student.lu.se

Michael Pater
agy15mpa@student.lu.se

June 14, 2021

Master's thesis work carried out at Robert Bosch AB.

Supervisors: Emma Söderberg, emma.soderberg@cs.lth.se
Ali Houmani, ali.houmani@se.bosch.com
Robert Lagerstedt, robert.lagerstedt@se.bosch.com

Examiner: Görel Hedin, gorel.hedin@cs.lth.se

Abstract

Program analysis is a great tool to improve code quality. However, problems such as false positives and complicated integration makes it less attractive to use. In this master's thesis we integrated the open source system MEAN (MEta ANalyzer) into the tool stack of a team at Robert Bosch AB. MEAN is a system that tries to alleviate some of the problems with program analysis via a data-driven approach. After MEAN had been running at Bosch for a few weeks, we deployed two improvements intended to reduce the amount of unwanted analysis results, in other words, noise that MEAN produced. Due to a limited time frame, the results were also fairly limited, but they show that MEAN is a promising system that software tool stacks could benefit from integrating. Thus, shifting the focus to more valuable code review with more efficient communication between tool maintainers and developers.

Keywords: Program analysis, MEAN, Robot comment noise reduction, Code review, Gerrit

Acknowledgements

First and foremost we would like to show our gratitude to Emma Söderberg for all the valuable help and feedback she has given us throughout this thesis. We would also like to thank the people at Robert Bosch AB for their warm hospitality with special thanks going out to our Bosch supervisor Ali Houmani for his help and guidance. Also a special thanks to the Bosch team in Austria that let us integrate the MEAN system into their tool stack. Last but not least, thanks to the interview participants that gave us invaluable feedback about MEAN.

Contribution Statements

Software Related Work

Throughout the thesis, both authors have discussed and planned the added software features and improvements. They have also worked on setting up, testing and integrating the MEAN system into the tool stack for a team at Bosch. Finally, some minor improvements such as some bugs have been found, fixed and pushed to the MEAN open source project by both authors, with commit hashes:

- 1d6ff547d880bfa8f658aaacb27f5c3b9eb6a593
- 6edec740e994b4416f676de94e9fb5a9ce7b908f
- 60c2c5dbd8c8d33280465afa2004e1e8ecea919c

Mattias has implemented the `Storage Publisher`, written initial local setup guide, implemented the wrapper for *CPPCheck* and Framework Analyzer, the python script that summarizes quantitative data into graphical representations, and lastly the `Filter Module` feature addition to MEAN.

Michael has added sections and improvements to the local setup guide, and implemented a wrapper for the Correct Mappings Analyzer at Bosch. A new feature to the MEAN system called `Visual Feedback Feature` was also implemented by Michael. Michael has also focused on testing and investigating proxy issues, security measures and compatibility of MEAN with an in-house Linux based OS.

Thesis Writing

This thesis has been written and discussions of the overall structure of the report have been conducted by both authors. They have also pair-wise written the Introduction Chapter and section 8.1 of the Results Chapter. Mattias has written the initial versions of the following chapters and sections: Abstract, Method Chapter, Related Work Chapter, The MEAN System Chapter, Integration of MEAN at Bosch Chapter, Mean Extensions Chapter (except section

7.4), section 8.2 of the Results Chapter, Discussion Chapter (except section 9.3), Threats to Validity Chapter and Conclusions Chapter. Michael has been responsible for formatting and designing the report structure. In addition to this, Michael has also written the initial versions of: Background Chapter, several additions to, and restructuring of, the Method chapter based on feedback, section 6.1 of Integration of MEAN at Bosch Chapter, MEAN Extensions (section 7.4), section 9.3 and parts of 9.4 of the Discussion Chapter.

Interviews and Networking

Both authors have been active during the interviews and communication with the focus team at Bosch. Mattias designed the initial invite to the team while Michael wrote the initial interview guide. Michael conducted the interviews while Mattias recorded the audio and took notes during the interviews. Michael transcribed the interviews.

Contents

Chapter 1
Introduction

Static program analysis is an important tool for development of high quality source code [1]. Even though it has been in practical use for some time, existing tools are burdened by usability problems [2–4]. For instance, false positives and incomprehensible results from the analysis tools reduce the value gained by their usage. For example, false positives arise when the analyzer incorrectly tells you that something is wrong. Take Listing 1.1 for example, this shows a simple code example in the programming language C where allocated memory is deallocated within a conditional statement. Since the condition will always evaluate to true due to the global value *staticTrue* is not zero, the allocated memory will always be freed.

```c
#include <stdlib.h>

static int staticTrue = 1;

int main(void)
{
    char * data = (char *)malloc(10*sizeof(char));
    if(staticTrue)
    {
        free(data);
    }
}
```

Listing 1.1: C code with a conditional Memory Leak

Here the issue arises when a static code analyzer does not realize this and incorrectly assumes that there is a memory leak. That is, the allocated memory is not deallocated before the end of the program, due to it only occurring when the conditional in the `if statement` evaluates to true.

To remedy this, meta analysis systems have been developed to gather data used to evaluate the usefulness of static program analysis [5–8]. A meta analysis system does not analyze code by itself but instead integrates and manages analyzers. By layering the meta analysis systems over analysis tools like in the above example, the system can then collect data. This data

can then be used in a process where feedback is given by developers, to the maintainers of the system or tool upholders. The latter can then respond to this feedback by configuring or possibly fixing problems if deemed valid. The changes are then made available to the developers where this cycle can then continue to repeat, thus creating a data-driven feedback loop for configuring the tools, as can be seen in Figure 1.2. Thus this meta analysis system integrates into the workflow of the developers and maintains a feedback loop between analysis tool maintainers and the developers that are using the results. MEAN (MEta ANalyzer) is one such system [8].

The result from the above example code in C can be seen in Figure 1.1, is presented by the MEAN system (detailed in Chapter 5), in the open source code review tool Gerrit (expanded upon in Section 2.3). This analysis result is in the form of a robot comment which is a type of comment generated by third party systems for Gerrit. The feedback loop mentioned earlier, starts with the blue 'NOT USEFUL' button seen in Figure 1.1. Pressing the button sends a message back to the MEAN system that this analysis result was not useful to the developer. The analysis tool the result was generated by can then be investigated to find out, for example, if the result is a false positive, not relevant to the project or unclear. Changes can then be made to the tool if the feedback is deemed valid.

Figure 1.1: A Robot Comment showing a false positive that would appear in Gerrit from MEAN for the above C code.

We performed the thesis study at Robert Bosch AB in Lund, a company known for their hardware ranging from power tools, dishwashers and home appliances to components in the automotive industry [9]. In recent years, the company has started focusing on software development as well, where the office in Lund develops software solutions. These are mostly parts of embedded systems related to the automotive industry but have also started working on electric bikes. An interest in smart applications relying on artificial intelligence has started gaining traction at the Lund office. Due to the increasing amount of software developed, the need for improving code quality has increased, which in turn connected this thesis with the Lund office. By improving the presentation and management of analysis results of software code, prospects are that the code quality will increase. Another aspect with favorable prospects is the value of development time will increase as a result of better configuration and feedback related to analysis results.

Thus, in this thesis, we study the company and the compatibility of the prospects of integrating MEAN into it. We then continue by setting up a functional version of MEAN, after which, we integrate it into to their tool stack. Concurrently with the integration, we implement improvements to the MEAN system, focusing on reducing the noise of presented

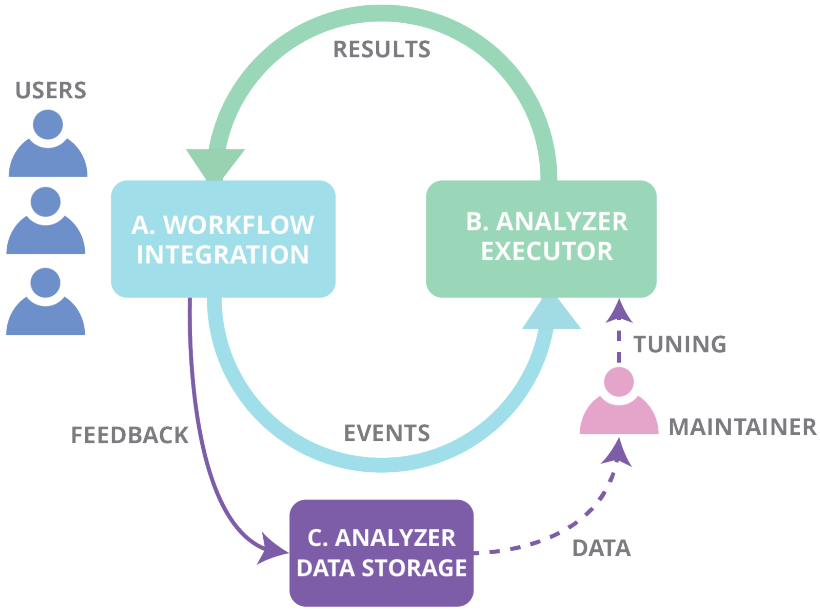

Figure 1.2: A meta analyzer feedback loop detailing the flow in the system. Users commit code in A which triggers events that cause analyzers to execute in B. Results are then sent back to the developers in step A. Users can give feedback based on the results, which is sent to C, where maintainers can use the feedback data to tune the system in B.

analysis results. Concluding these steps, we perform interviews with developers exposed to the changes we made, to gather qualitative data. We complement this data with quantitative data about the program analysis events generated by the MEAN system by gathering it in a database.

1.1 Objectives

The goal of this thesis is to integrate MEAN into Bosch's workflow and tool stack. We also want to evaluate, from a cost-benefit perspective, how the feedback and integration gained by using MEAN affects the usage of analysis results. The overarching objective is therefore to answer the question *"What is the effect of integrating MEAN into the developer workflow at Bosch?"*.

In this thesis, we expand on the work of an earlier thesis that explored meta analysis and took place at Axis Communications in Lund, Sweden [10]. The authors of the related thesis implemented and integrated the open source software called MEAN [7, 8] at Axis Communications, introducing robot comments into Gerrit [11] reviews. Thus one of our objectives in this thesis is to explore the possibilities of integrating the MEAN system into another company's development environment, in this case Robert Bosch AB in Lund. Ljungberg et al. [12], found that program analyzers that flooded the developers with noise through superfluous amounts of results were opted out. This then limited the usage of valuable analyzers since they hampered the developers work. We thus aims to explore the value gained by reducing the noise by improving the MEAN system with said functionality.

1.1.1 Research Questions

We have formulated the following research questions from the above objectives of this thesis:

- **RQ_1** How can the architecture of the MEAN system and its integration into Gerrit be improved to reduce robot comment noise?
- **RQ_2** How can MEAN be integrated into Bosch's workflow and tool stack?
- **RQ_3** How do developers at Bosch perceive the value of analysis results?

1.2 Delimitations

This master's thesis is limited to the integration of the MEAN system for a selected team at Bosch. This means that MEAN should be able to send robot comments to Gerrit, as well as gather data about the responses, for a select set of program analyzers for the chosen team at Bosch.

Due to the pandemic (COVID-19) during the execution of this master's thesis, the majority, if not all, of the work we perform, is limited to a virtual environment. Thus we will have to conduct the interviews in digital form.

1.3 Risks

This section presents several risks with the project and planned mitigation of those risks.

1.3.1 Difficulties with MEAN integration

Depending on the different systems that Bosch uses, smaller or larger changes will have to be made to MEAN before integrating it into the tool stack. This can cause **RQ_2**, concerning the integration of MEAN into Bosch, to be compromised in the level of detail of the answer.

Mitigation We will try to learn about the system at Bosch as early as possible, to be more prepared for the changes that have to be made to MEAN when the time for integration has come. Mindfulness in regard to potential increase in integration time will be reflected in our schedule.

1.3.2 Not enough quantitative data

To be able to answer **RQ_1** and **RQ_3**, a fair amount of data is needed from the MEAN system.

Mitigation To mitigate this risk, we will follow a strict deployment schedule to make sure that MEAN is deployed early enough to gather a sufficient amount of data. Also, a weekly meeting will serve as a checkpoint to verify that everything is moving according to the plan.

1.3.3 Not enough qualitative data

The accuracy of representative data we will have gathered from interviews is heavily relied upon by $\mathbf{RQ_3}$, as it related to how developers perceive analysis results. Another risk arises when considering the fact that we will be conducting this work remotely, thus limiting day to day social interactions with the developers at Bosch.

Mitigation We will lessen the severity of the first issue by conducting interviews with several different developers of different roles. While the second issue can be mitigated by good planning on our side and coordination with team leaders. A detailed email survey can instead be used to alleviate the limited social interactions with developers. With the help of team leaders, we can hopefully ascertain participation from employees with the email surveys.

1.3.4 Data Restrictions

A potential risk that can impair the gathered data can arise due to security restrictions that may be attributed to some of the collected data.

Mitigation Early meetings with involved security administrators and supervisors can help specify what data is restricted and what alternate methods can be used.

1.3.5 Infrequent Developer Activity

Another risk that might limit the amount of data available can arise when code changes are infrequent. This would mean that there would be fewer analysis results, and in turn less feedback back to MEAN.

Mitigation Since it is out of our control to change the code change frequency, the solution to lessen this risk would be to have a large amount of analyzers integrated and enabled in MEAN. This would mean that even if the code changes are infrequent, there would be a large amount of results when they eventually occur.

1.3.6 Low User Engagement

Even if there is a large amount of analyzers integrated into MEAN, which normally produces a lot of analysis results, there is still a risk of little feedback from the developers. This could occur if a large portion of the analyzers are already in use at Bosch, and are thus already 'run in'. This would mean the analyzers would produce fewer results than otherwise expected. Even when they produce results, they are likely to be relevant results, and therefore less 'NOT USEFUL' feedback would be sent back to MEAN. Another cause of low user feedback may be due to the risk of developers being unfamiliar with the system. If changes appear without any explanation to them, then users may be vary of them, or not even notice them.

Mitigation By selecting some analyzers to integrate into MEAN that have not been previously used in Bosch, and are likely to produce more noise, we hope to alleviate the above mentioned risk. In addition to this, information regarding the functionality of the system will be sent out by us to the team affected by our changes. These will be an email, a post in their teams channel, a small wiki page as well as a video demonstration.

Chapter 2
Background

This chapter gives a brief overview of the components that MEAN integrates with and use. Many of these components are based on and used for, continuous integration (CI). Thus this development strategy and the software tools that are used in this thesis are explained first. This exploration is then followed by the other components MEAN connects to, or uses. Some of which are containerization, a message queuing system and program analysis tools.

2.1 Continuous Integration

Continuous integration is a development practice in software development, especially common in the agile variety. The focus of this practice is to minimize bugs and problems that may arise due to merging repositories [13]. The main problem arises when developers clone a code base to work on a specific feature, a bug fix or other software related issue. The longer the developer waits to merge their work back into the code base, the bigger the changes may be to this main code base. Thus conflicts due to the merge may arise and lead to tedious and time consuming work being put into solving the issues.

The practice of continuous integration aims to commit changes often and thus smaller changes are integrated each merge. This simplifies the detecting and searching for issues, as the new code will be more congruent with the main code base. Since the change is smaller, it is therefore easier to pinpoint where any issue may lay if such problems do incur.

To facilitate this practice, tools are used to automate integration by building and testing as well as tools that maintain a code base. Such code base related tools may then allow for the review of the code and changes to be made in a more controlled and intuitive context.

A typical situation where a CI system is triggered is shown in Figure 2.1 where the steps are detailed below:

1. The developer makes a change and commits it to the repository.
2. The CI tool triggers on the commit and fetches changed code.

3. The CI tool builds and tests the fetched code.
4. If successful, the result from the build may be deployed/released, generally automatic in the step above CI, Continuous Deployment (CD).
5. If not successful the team is notified of the problem so that they can fix the issue and try again.

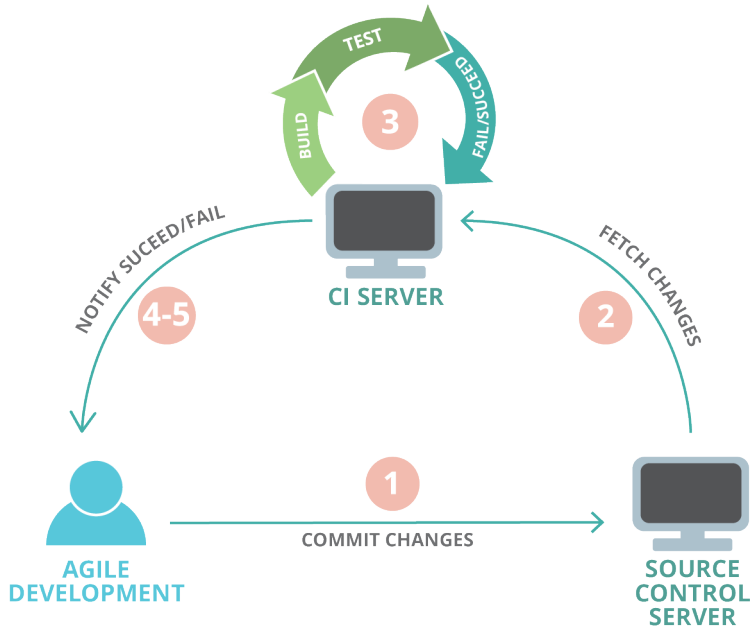

Figure 2.1: A graphical representation of a typical scenario where a CI system is triggered.

There are tools available, capable of handling human, error-prone situations such as testing and building of code with the new changes in place. As such errors can otherwise break the entire code base and cause severe issues that most likely will consume valuable developer time. The same is applicable to testing and building as such tasks in modern software development usually entail a large number of transitions between stages. In the next step of CI, namely Continuous Development (CD), the changes that are built successfully by the automation server are then deployed as artifacts directly to the product [14].

Thus, automating these stages so that the changes are tested and built in a safe environment is a major benefit to developers. The automation is usually performed by a core, also known as a master, managing several agents, also known as slaves that do the work. The more agents you add the more changes can be built concurrently. Which is important to consider since builds in modern software development can often take several hours.

Jenkins is an automation server used for CI/CD by utilizing pipelines governed mainly by scripts [15]. It is a free open source, java-based software, widely used in modern software development. Jenkins has a vast number of plugins that enable flexibility in designing builds and is thus easy to integrate into the development workflow.

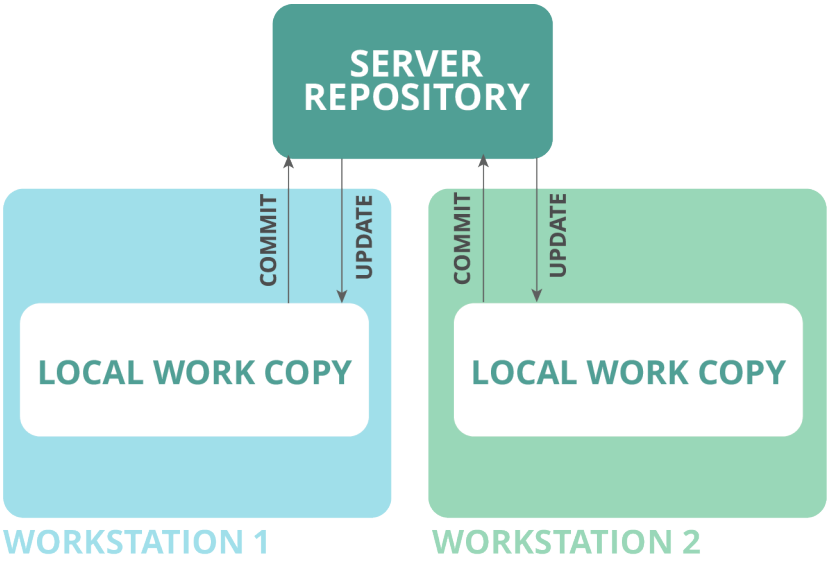

Figure 2.2: An overview of a centralized VCS.

2.2 Version Control

To be able to keep track of the changes made to the source code, a version control system (VCS) can be used [16]. This greatly simplifies management of multiple developers making changes to the same code base. Thus, pinpointing errors caused by a change is easy to locate and solve.

Branching helps developers make changes concurrently to the same code base since they all have a local copy of the code. Changes made by developers are kept track of by the VCS. Merging the code to the main source will then be logged for when and whom made what changes. In addition to this, if changes interfere with each other when merged, the VCS will systematically locate this issue and allow the developers to solve the conflict. In some cases the VCS can even solve merge conflicts automatically.

Due to this, VCSs are invaluable to agile software development, where changes made by developers need to be merged often and many times in a day. An overview of a typical version control system can be seen in Figure 2.2, depicting a centralized model. For this type of model, the central repository contains the history of changes made. Here all developers share one main repository and as such, are dependent on the main server to jump between versions.

Git is a version control system that is among the more popular tools used in the software industry [17]. It is also the VCS used in the environment this thesis is done in. Among the different kinds of VCSs used, Git is a distributed VCS. In a distributed VCS each copy of the main source a developer is working on is its own working repository. Thus, this local repository will also contains the history of changes made, as can be seen in Figure 2.3. Thus if the main server goes down, a developer can still work with the assurance that the changes committed will be kept track of.

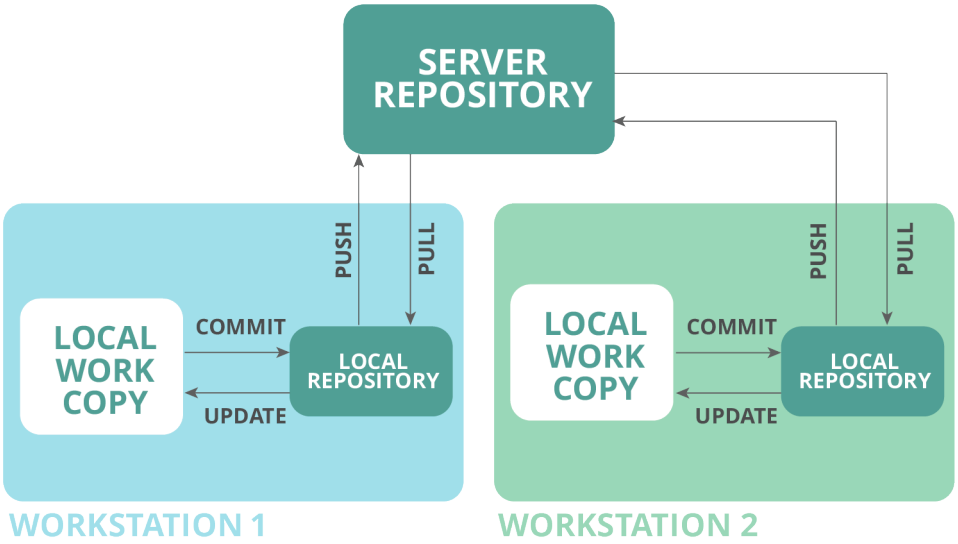

Figure 2.3: An overview of a distributed VCS such as Git.

2.3 Code Review

There are always bugs that arise when coding due to our human nature. Variations of coding styles can bring frustration to other developers trying to understand the code. Thus code practices and policies are utilized in development teams, let alone companies. Such guidelines helps maintain the readability of the code in a team. Code review is a well-established concept with benefits such as finding defects, making team level coding conventions easier to maintain as well as help maintain a shared authorship of code written by team members. These advantages are detailed by A. Bacchelli and C. Bird, which state that the mentioned aspects have greatly increased in use with the help of tools [18].

When a feature, bug fix or other change has been completed, the developer will publish it for code review where other developers will review it. If any problems are found then the reviewers will mark the code for revision with comments on the issues. The developer will then be tasked to fix these to have their code added to deployment. When a revision has been made, another session of review is performed. This iterative process then continues until the reviewers give a pass for the code.

In a case study at Google, Sadowski, C. et al concluded that code review enables knowledge sharing between developers, helping the team as a whole improve, as well as improve the learning phase for junior developers [19]. Hence reviewers can learn new techniques from the developer that submit the changes. The submitting developer, may on the other hand learn from the critique given by the reviewers.

Code review is done before the code is truly submitted to the version control system. This helps bugs and other issues to be found early in the development cycle. As such, this greatly saves time and effort when compared to being found later in development. Improvements to the code can be commented on by the reviewers as well, thus highlighting the knowledge sharing in this practice.

Gerrit Code Review is an open source code review tool with Git at its core [11]. It automates code review and version control related procedures. Examples of which are: creating

a branch for a change and supporting in-line code comments from software. At the same time, Gerrit offers an intuitive platform for the reviews to take place. This is done by acting as a channel between the commits from developers and the main code base repository. It automatically creates review branches for each new change to be committed. By amending commits the changes can go through iterations, also referred to as patch sets. The reviewers can then, at each patch set, decide at their own leisure whether the change is acceptable or not.

With the help of comments and numeric scoring, both from humans and automatically for instance, by continuous integration, communication can be done between the developer and the reviewers. The integer based scoring labels range from +2 to -2, the meaning of each can be seen in the list below [20]. A change must have at least one +2 with no votes with -2 labels to be accepted. It is also important to note that the labels do not accumulate. An overview of the Gerrit system and connecting components is shown in Figure 2.4.

+2 "Looks good to me, approved"

+1 "Looks good to me, but someone else must approve"

0 "No score"

-1 "I would prefer that you did not submit this"

-2 "Do not submit"

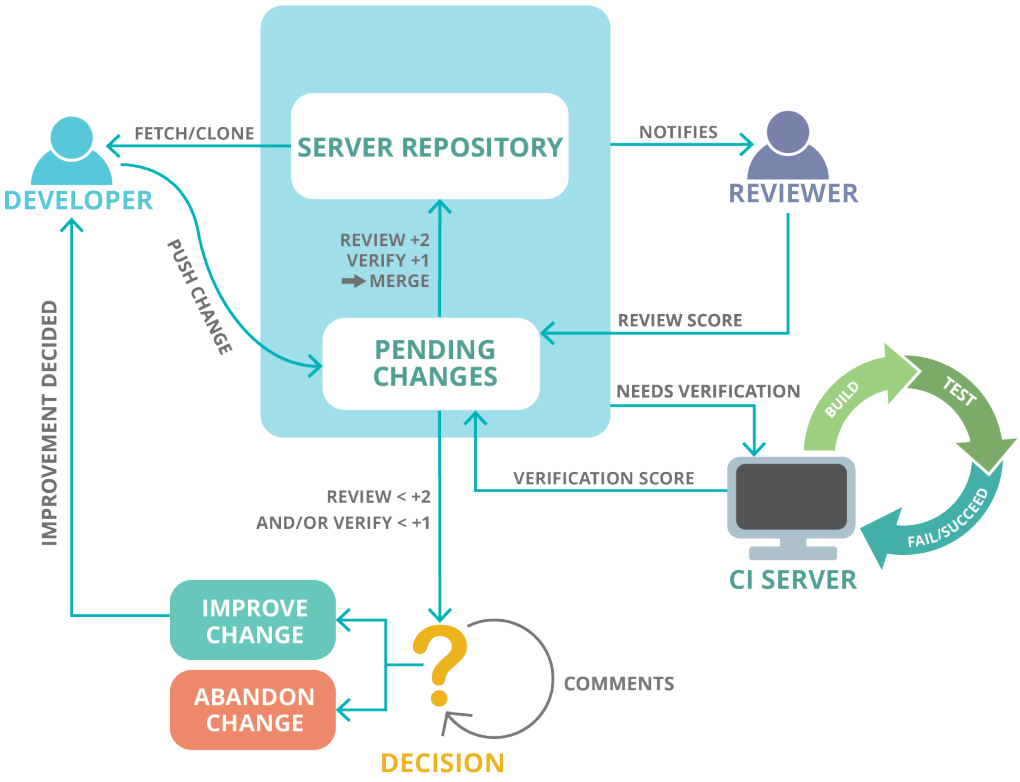

Figure 2.4: An overview of the relationship between reviewer, CI server and developer tied around Gerrit.

2.4 Containerization

Virtualization at the operating system level can yield several benefits to development and execution of software. The two common virtualization techniques are virtual machines (VMs) and containers, sharing many attributes. Containerization is a modern and lightweight approach to virtual machines, greatly reducing the high resource overhead otherwise instilled. In both cases, software is isolated together with the libraries, binaries, configurations and other necessary dependencies to run the application. All running on top of the operating system of the host machine.

The main difference between virtual machines and containerization is the extra overhead induced by each virtual machine. Since each VM needs its own virtualization of the host such as hardware, CPUs, the stack and other necessities of running the application, as can be seen in Figure 2.5. Then they need to access the host operating system and resources via interfacing through a hypervisor further incurring overhead [21]. Containers instead make use of the host machines kernel, and as such all container share the same kernel as can be seen in Figure 2.6. The containers are thus smaller than VMs, where each instance has its own virtualized kernel. These containers then interface to the host operating system and its resources via a runtime engine. Due to this, the containers are faster and less resource intensive to run. Thus, several containers can run on a host where one virtual machine would struggle. The main benefit of containerization is isolating the software from the operating system. This adds high portability benefits which eases development time of these small software packages. Compatibility issues between machines are thus removed [22].

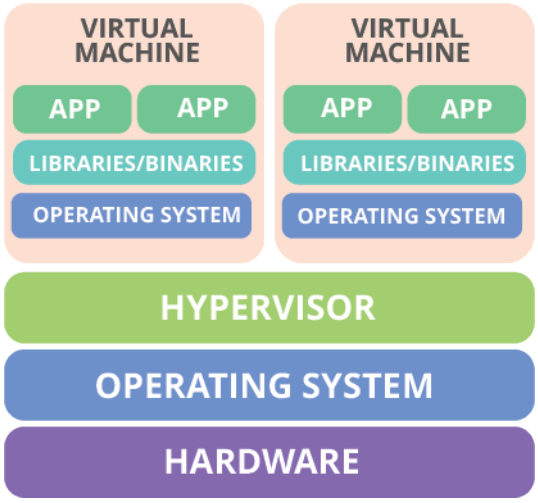

Figure 2.5: Overview of how a virtual machine is structured upon the host. Notice the need for a virtual kernel in each virtual machine.

Docker is the leading platform for working with containerization, and helps build and run containers [23]. Written in the Go programming language, it has both a premium edition and a free open source community edition.

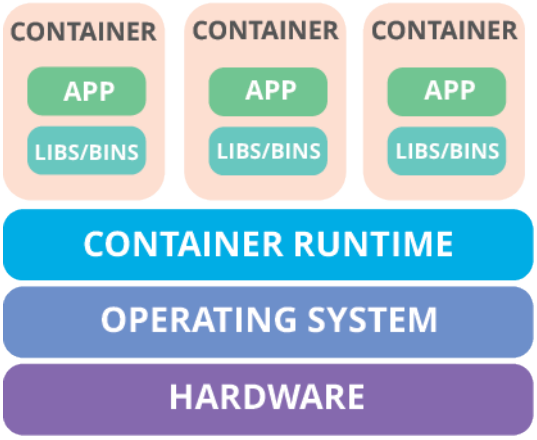

Figure 2.6: Overview of how a containers being run on top of the host. Notice the there is no need for a virtual kernel in each container.

2.5 Message Communication

Messaging techniques can be used as a method in cases when software components need to communicate with each other asynchronously. Using queues to contain the messages allows components to perform other tasks instead of waiting for a response. Messages are generated by a producer, which are then sent to the message queue and stored in order. These messages are then consumed by another component, thus fulfilling the communication without the need for both parties to have been "present" simultaneously.

Once a message is consumed it is deleted and thus removed from the queue. This can cause problems if several applications need to "read" the message sent from another unit. To accommodate this problem, a publish-subscribe methodology, visualized in Figure 2.7, is used to allow several components to "subscribe" to a queue. A message will not be removed until all subscribers have "read" the message.

RabbitMQ is a lightweight, open source message broker with a wide variety of features, the publish-subscribe method is one among them [24]. It supports several message protocols, such as the Advanced Message Queueing Protocol (AMQP).

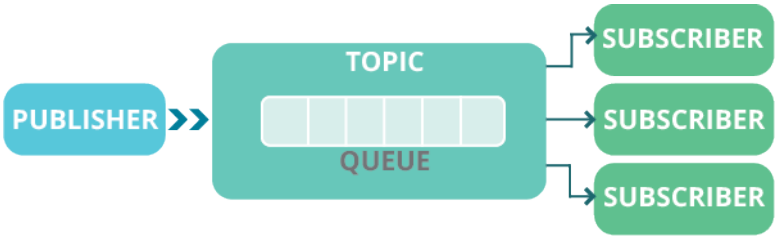

Figure 2.7: An overview of a simple publish subscribe relationship. There can be one or more publishers, generating messages to a queue tied to a topic. Multiple subscribers can sign up for the queue of a specific topic. Once all subscribers have read a message in the queue, it is deleted.

2.6 Program Analysis

Program analysis is used to maintain reliable and bug free software code. It can even help optimize the code for improved performance and memory usage. There are two categories of program analysis, static and dynamic. In dynamic program analysis, the code is analyzed while the program is running. Examples are debuggers, bench marking tools, and unit tests. On the other hand, static analysis is done without running the code. It is usually done before or during compilation of the software code. Examples of static analysis are type checkers, optimizing compilers, and static checkers.

In this thesis, we focus on the usage of automatic static program analysis tools. Thus the results can easily be collected after having run the tools on the presented code by use of wrappers, which will be further described in 7.2.

Cppcheck is among the analyzers used in this thesis. It is used to analyze and find issues such as memory leaks and bugs in C and C++ code [25].

Chapter 3
Method

This chapter outlines the steps we took to complete this project. With our goal being to find the answers to the research questions formulated in Section 1.1. The path taken to find each answer is detailed in Figure 3.1, showing relevant components of this chapter for each research question. Our first involved reviewing related work by performing a literature review (method described in Section 3.1.1) to form a basis for $\mathbf{RQ_1}$, regarding how to reduce robot comment noise, and $\mathbf{RQ_3}$, regarding how the developers perceive the analysis results. In this step, we also examined the MEAN system (method reported in Section 3.1.2). As such, we could establish the relevant understanding to undertake the next steps for all three research questions.

Having studied and accustomed ourselves to the architecture of the MEAN system, we began the integration process (method reported in Section 3.2). We started with adapting it to the tool stack used at the Lund Bosch office. We did this by setting up a local version of the MEAN system within the Bosch network. During this step we also made some minor bug fixes to the MEAN system source. In addition to this, we also added an essential component for quantitative data gathering to the running local version of the system (also described in Section 3.1.2). Following this we were able to easily switch over to the main production line with some minor changes in configuration. As such, we deployed the MEAN system to the software team in Austria (method described in Section 3.4). This gave us the necessary understanding to answer $\mathbf{RQ_2}$, how to integrate MEAN into Bosch, as seen in the middle path in Figure 3.

None of the already existing wrapped analyzers in the MEAN open source where usable due to not being relevant. We thus had to implement three new wrappers to take on the task of producing analysis results for the Austrian team (method reported in Section 3.3). In addition to this, based on related work, we implemented two noise reduction features, a `Filter Module` and a `Visual Feedback Feature` (method described in Section 3.3) with the focus of answering $\mathbf{RQ_1}$.

To be able to answer $\mathbf{RQ_1}$ and $\mathbf{RQ_3}$, we gathered both quantitative (produced and collected by the MEAN system, method described in Section 3.5.1) and qualitative data (gathered

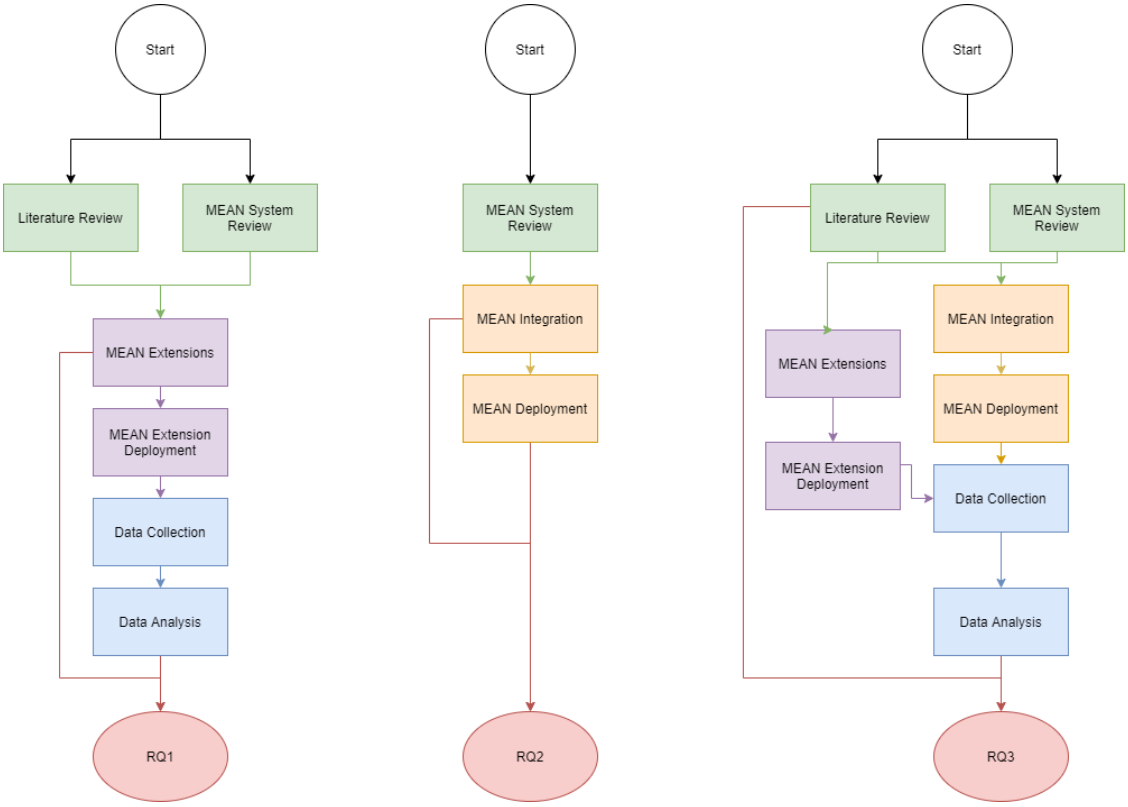

Figure 3.1: An overview of the different steps taken in this thesis to answer the proposed research questions. Each flow graph shows the path taken from start to an answer for each research question.

through interviews, the method of which is reported in Section 3.5.2) both methods found in Section 3.5. We finished with analyzing the gathered data (method reported on in Section 3.6), thus reaching the end of the path as shown in Figure 3.

3.1 Review Related Work

In this section we describe how we reviewed related work by performing a literature study and exploring the architecture of the MEAN system in depth. Finishing with a study of the context of our thesis compared to the previous one.

3.1.1 Literature Study

Early in the project, we conducted a literature study. We did this by searching through the academic database LUBSearch, as well as going through the references of a few articles [2, 3, 5] given to us by our university supervisor. As can be seen in Figure 3.1, the main reason for conducting the literature study was for us to get an understanding of how earlier work in meta analysis was done. As well as the finding out what shortfalls and future work was described. With this knowledge, we had the necessary means to answer research question about reducing noise, $\mathbf{RQ_1}$. The means of which, was formulated by assessing which ways we could improve

MEAN to reduce robot comment noise. In another part of the literature study, we focused on research done on the usability of program analysis. This gave us an understanding of why meta analysis is needed in the first place. We detail the results of the literature study in Chapter 4.

3.1.2 MEAN System Review

To be able to install, set up and run the MEAN system, we first needed to understand how it worked. Thus, we studied the MEAN source code in detail to form an overview of the architecture of the system. To better help us understand the design choices made, we also relied on the thesis report of the developers of MEAN [7]. As such, we formed a cohesive understanding the MEAN system which we describe in detail in Chapter 5.

We then put this knowledge in practice to further form an understanding of the system. We did so by getting all the components that serve as the MEAN system to run on a local machine. During this experimental phase we discovered some minor bugs in the code, which we fixed. Later, we added these fixes to the source code via a process of reviewing by the emerging MEAN open source software (OSS) community. In addition to this, we kept detailed notes of the method performed to get the MEAN system to run locally. A guide to set up and run the MEAN system locally was then written by us from these notes and later added to the above mentioned repository.

We also discovered the lack of a storage module to collect and store the quantitative data produced by the MEAN system. As such, we implemented this storage feature (presented in Chapter 7). The module collected and stored the messages sent via RabbitMQ, in the MEAN system. Examples of which are: analyzer events, robot comments and 'NOT USEFUL' presses, all of which were stored in a MongoDB database. MongoDB is a document-based database system that stores entries in JSON format. This module was later added by us to the MEAN OSS through the same process the bug fixes went through.

3.1.3 Study Context

Based on a previous thesis where the MEAN system was built and integrated at Axis Lund, we formulated **RQ$_2$** to explore the process of integrating the MEAN system at another company. Thus, we would gain valuable insight in increasing the accessibility of MEAN. For this purpose, Robert Bosch AB in Lund was selected by us due to their interest in the MEAN system and the differences between it and Axis.

In the software department, within hardware development, Axis had over 1150 developers while there were, at the time of this writing, only about 200-300 developers at the Bosch Lund site. But due to the strict regulations regarding adding plugins to the production Gerrit at Bosch Lund, we could not integrate the MEAN system there. Instead, we found a team through the help of our Bosch supervisor of roughly 50 developers in Austria. We found this team since they had full control of their tool stack. As such, our thesis greatly differs from the previous thesis at Axis in number of software developers exposed to MEAN.

While our work took place at the Lund office, we integrated the MEAN system in the tool stack of the team in Austria. We found that mainly in-house tools were used at Bosch, developed for specific purposes. The analyzers used in the Austrian team were built by single developers. In contrast, Axis housed a large team dedicated to developing tools, when comparing a team from Bosch with the entire Lund office of Axis.

3.2 MEAN Integration

We integrated the MEAN system in two stages: setting up a local running version and connecting it with the production tool stack of the selected team. The first stage used what we learned from the review of the MEAN system (see Section 3.1.2) where we followed the detailed step by step guide we wrote to set up theMEANsystem. Thus, we installed and ran the system on a computer provided by Bosch that was connected to their internal network. As such, we started of with this local test environment for the first phase of the integration. Since the system would be running within the Bosch network, the transition needed to switch over to the production tool stack could be quickly performed by us. As this only involved minor changes to the configuration of the MEAN system. Our main reason for the first integration phase was thus to work out a few Bosch-specific uncertainties, without the risk of disrupting the production system for the developers.

While setting up the MEAN system locally, we discovered the lack of RabbitMQ at Bosch which resulted in us having to research possible message brokers to use (reported in Section 6.2). In the end, we decided to set up a small RabbitMQ server for the purpose of our thesis. At the same time, our usage of Docker and the building of images was impacted by strict proxy restrictions set up by Bosch for security measures. To solve this problem, we studied Docker and Bosch related Docker guides as well as spoke with developers experienced in such matters. As a result, with some minor modifications we managed to modify the Dockerfiles such that we could build the images in harmony with the proxy.

Meanwhile, we slightly modified the MEAN Gerrit plugin to be compatible with the somewhat older version of the Austrian team's production Gerrit. Once done, we built the plugin and installed it into their Gerrit, hibernating for when the MEAN system was activated.

Before we could deploy the system in a state where analysis results would be produced, we had to study the static code analyzers used by the team. As none of the available analyzers provided by the MEAN system were relevant to the team and its software. Once we had found valid analyzers had and integrated them into the MEAN system (method reported in Section 3.3.1) we could deploy the system (approach detailed in Section 3.4.1).

3.3 MEAN Extensions

This section details the additions we made to the MEAN system. With Section 3.3.1 detailing the process of studying the Austrian teams active analyzers. Followed by the wrapping and finally incorporating them into the MEAN system. While Section 3.3.2 explores how we decided upon the two extensions for the purpose of finding answers to $\mathbf{RQ_1}$, how to reduce robot comment noise.

3.3.1 Wrapping Bosch Analyzers

We were unable to use any of the already available analyzers wrapped and packaged in the MEAN system. This was due to the fact that none of them where applicable to the software of the Austrian team, reasons of which are explained in Chapter 8. As such, we discussed with a team responsible what analyzers they used for which we could wrap and incorporate into the MEAN system running on our Bosch server. What we discovered was that they mainly used

in-house program analysis in their production line. As such, we chose two different tools to wrap (reported on in Section 6.1). The main reason for this was that they produced the most results, and as such, compensate due to the small size of the team.

The first analyzer consisted of about 50 modules, none of which adhered to a standardized output scheme. This meant that we needed to select the most useful, in other words, most noisy, plugins so that wrappers could be created for them. As it was unfeasible to wrap all available plugins within the time constraints of our thesis. The second one only produced one result per build for an entire project, if certain conditions were met.

Later, we chose an 'off-the-shelf' analyzer to be added due to issues explained in Section 3.4. This analyzer was Cppcheck [25] (see Figure 6.1 for an overview of MEAN that shows all the wrappers), and involved a relatively painless process of creating a wrapper for.

3.3.2 Noise Reduction

We devised two 'intent-to-implement' proposals based on what we learned from the MEAN architecture (detailed in Chapter 5). After which, we combined them with related work (described in Chapter 4) to answer $\mathbf{RQ_1}$ about noise reduction. These proposal went through a review process in the emerging OSS MEAN community, resulting in our solutions described in Chapter 7. The first was a feature that filtered out analysis results between reviews that were not generated from changed lines. The second one added a `Visual Feedback Feature`, altering the color of robot comments based on 'NOT USEFUL' clicks to visually mark sticky robot comments.

3.4 MEAN Deployment

We divided up the deployment into two parts, as shown in Figure 3. In the first step, we deployed a vanilla system consisting of the fully integrated MEAN system with added analyzers (repported on in Section 3.4.1). Following this step, we performed a feature update (detailed in Section 3.4.2) that added the noise reduction extensions previously mentioned.

3.4.1 Vanilla Deployment

After we had implemented a working version the wrappers for the first analyzer group, we finally deployed the MEAN system. This was done by re-configuring the MEAN system to instead communicate with the Austrian team's production Gerrit and test Jenkins server. As such, Bosch developers of the Austrian team started to receive robot comments in Gerrit with analysis results. Two work-days later, we had completed and finished testing the wrapper for the second analyzer, deploying it as well. Barring a few bug-fixes, we made no improvements to MEAN during this phase. As such, the system was behaving roughly in the same way as it did for the original users of MEAN.

After a week, we had observed a low amount of analysis results produced by MEAN. We had suspected this risk of happening, as we've written in the risks Section 1.3.5 and Section 1.3.6. Thus, we opted to include a more noisy, 'off-the-shelf' analyzer to retroactively mitigate this risk. This third and last analyser was Cppcheck [25], (see Figure 6.1 for an overview of MEAN that shows all the wrappers). Before we enabled it, we first ran Cppcheck

on one of the repositories in Gerrit. As such, we could see that it would actually produce any real amount of results, which it did. We selected Cppcheck specifically because it was an 'of-the-shelf' tool that might garner more 'NOT USEFUL' feedback, unlike the two more finely tuned in-house analyses.

What we learned through the study of the MEAN system followed by integration and deployment into a Bosch teams tool stack was then used to form an understand to which we could use to answer $\mathbf{RQ_2}$.

3.4.2 Extension Deployment

Three weeks after the deployment of the vanilla system, we rolled out two improvements to MEAN regarding reduction of robot comment noise. Our original plan was to roll these improvements out separately, but due to time constraints, we decided to deploy them simultaneously.

We rolled out the `Filter Module` quietly by modifying the Jenkins script to accommodate the new module and then running the micro-service alongside the other MEAN components. On the other hand, for the `Visual Feedback Feature`, we first updated the local source code of the `Robot Publisher`, building and then executing it together with an update performed to the Gerrit plugin.

3.5 Data Collection

To be able to answer $\mathbf{RQ_1}$ and $\mathbf{RQ_3}$ we needed to collect both quantitative and qualitative data. The quantitative data was gathered with the help of the storage module (detailed in Section 7.1), further described in Section 3.5.1 below. We then gathered qualitative data by interviewing developers of the Austrian team, the process of which we detail in Section 3.5.2.

3.5.1 Quantitative Data

As mentioned earlier, the quantitative data consisted of the communication within the MEAN system as well as published robot comments and any registered 'NOT USEFUL' clicks. Having implemented a storage module that listened to these topics, saving them in respective MongoDB collections, the gathering of this data was performed automatically throughout the deployment of the system.

3.5.2 Qualitative Data

From what we discovered from our review of related work together with the experiences of the integration and deployment of the MEAN system, we designed an interview guide, outlined in Appendix 11. This guide was then used as a starting point when we gathered qualitative data by performing, loosely, semi-structured interviews [26] of varying lengths.

Originally we had planned to perform two sets of interviews, one each for the vanilla and extension deployments. The first set was to gather an overall picture of the tool stack of the Austrian team and how they perceived the MEAN system. Instead, in the second

set we wanted to focus in depth on the noise reduction improvements. The low amount of Gerrit activity and late deployment of the MEAN system for a small team, affected our originally planned interview approach. Through discussions with a responsible for the team in Austria we found that the time available for 30-minute-long interviews was infeasible. Another discovery was that only a portion of the team, located in Asia, had mainly been active with the production Gerrit. The team responsible also advised us to not use questionnaires as there was a very low participation rate for non-mandatory questionnaires. We therefore opted out of this approach since it was not feasible for us to ask for such mandatory surveys for the team. We thus concluded, due to the time constraints of this thesis through extensive discussions with the team responsible and help from our supervisor to relax the length of the interview time. This then gave us the possibility to perform shorter interviews, extracting the most important parts of the semi-structured interview guide originally designed.

Our main focus was then shifted to gather the opinions of developers that had interacted with the MEAN system. With the help of collected 'NOT USEFUL' clicks, we were able to target specific developers. Thus, we established a list of four developers with coordination from the team responsible, one of which had interacted with the 'NOT USEFUL' button. For the others, we presented some examples to thus still be able to gather feedback about the changes we made. During the interviews, we emphasized our willingness to hear any negative feedback to avoid participant response bias that can arise when presenting a tool you built to someone (discussed more in Section 10.2). Following the first interview, we managed to find an opportunity to perform a longer interview with the team responsible. We adapted it to a key-informant interview [27] based on the interview guide. In this specialized interview, we focused on the usage of analyzers in Bosch and the history behind the process of implementing and tuning their in-house analyzers. The second author performed the interview with the respondent, while the first author focused on taking notes and presenting the examples. This way, if the participant did not want to be recorded, or if the recording would be lost/corrupted, we would still retain the important data.

3.6 Data Analysis

We implemented a tool that read the collected quantitative data and produced graphical representations found in Chapter 8. To analyze the qualitative data, the second author transcribed the recorded audio of the interviews. These transcriptions where then analyzed together with the graphical representations of the quantitative data to formulate the results found in Chapter 8 and as such, used to find answers to $\mathbf{RQ_1}$ and $\mathbf{RQ_3}$.

Chapter 4
Related Work

This section briefly describes previous research and work done related to our master's thesis, in the area of program analysis and meta analysis. We found the related articles by starting from a few articles provided by our supervisor. With more added by us when we systematically went through their references to find more related articles. Also, we used the Lund University Libraries database (LUBsearch) to find articles. To find articles in LUBsearch, we used keywords such as 'static analysis' and 'program analysis'.

4.1 Program Analysis Challenges

Several challenges exist regarding the usage and integration of program analysis, which may deter developers from using such tools. In a paper by Imtiaz et al. [3], 280 Stack Overflow questions related to static program analysis were categorized to understand the challenges developers face when using static program analysis. They found that a large part of the questions were related to Ignoring/Filtering alerts (23.9 %), false positive validation (22.9 %), and how to fix the alert (19.6 %). A large part of these questions did not have an accepted answer, which indicates that the answers given were unsatisfactory. They conclude that static analysis tools would benefit from alert filtration and customization options. Meta analysis tools, in this case MEAN, is the tool we used in this project to try and reduce these problems.

Johnson et al. [2], carried out interviews with 20 developers to find out the attitude of developers towards static program analysis. Each interview was divided into two parts, where the first part consisted of questions related to usage and opinion of static analysis tools. The second part consisted of observing the participants while they used a static analysis tool, to get an understanding of the developer workflow. The results show that developers perceive a number of problems with static program analysis, which might prevent them from using such said tools. Among the interviewed people, 14 of the 20 pointed to poor output, such as an abundance of false positives and large number of warnings, as a hindrance. Customization of the analysis tools were another important aspect, mentioned by 17 participants. Another

important point was workflow integration of the tools, which 19 of the 20 people interviewed mentioned. In our project, MEAN is used to solve some of these problems. For instance, one purpose of MEAN is to make the integration of analysis tools easier.

Nachtigall et al. [4], describes usability issues with static program analysis. This was done by surveying research from the past decade to group usability issues into six different categories: 'Understandable Warning Messages', 'Fix Support', 'False Positives', 'User Feedback', 'Workflow Integration', and 'Specialized User Interface'. These categories were then used to explain the shortcomings of 14 state of the art static analysis tools, to see if and how the issues were addressed. By studying the 14 static analysis tools, three overall weaknesses were revealed: Too generic warning messages, false positives and limited user feedback, and developer workflow disruptions. Nachtigall et al. suggests that the creation of interactive systems that perform analysis based on user input could be used to enhance explainability. In our project, the meta analysis tool MEAN works as such an interactive system, which gives the developers the possibility of sending feedback back to the system.

4.2 Data-driven Deployment of Program Analysis

A meta analysis system is a system which integrates and evaluates program analysis tools. One such system, called Tricorder, was developed at Google [5]. The goal of Tricorder was to create a scalable and easy to integrate static analysis platform that would create a feedback loop between analysis creators and developers. Tricoder is built on five principles: *No false positives*, *Empower users to contribute*, *Make data-driven usability improvements*, *Workflow integration is key*, and *Project customization, not user customization*. To achieve the goals, Tricorder is implemented via micro-services, where each service has one specific task. New analyzers can be easily added to Tricorder, in part due to the modularity of the system. Every analyzer can be triggered to run in one of three different stages, FILES, where analyzers only know which source code files have changed, DEPS, where analyzers know about dependencies, and the final stage COMPILATION, where the analyzers have access to the abstract syntax tree of the program. This means that simpler analyses that can run in the FILES stage, can run earlier and provide faster results. The results of the analysis are then sent to the code review system as robot comments, and reviewers can mark the comments as "NOT USEFUL", "PLEASE FIX", "PREVIEW FIX", or "APPLY FIX". This feedback can then be used to tune or even disable analyzers. The number of code violations reported by the analyzers reduced over time, due to tuning of the system based on the feedback gathered via the robot comment buttons.

Another program analysis system, called Review Bot, was developed by VMware, to automatically integrate static program analysis into their code review process [28]. Review Bot integrates three Java analysis tools (PMG, Checkstyle, and FindBugs). However, the system is built in a way that makes extending it with analysis tools for other programming languages possible. Unlike Tricorder, Review Bot has to be manually invoked, by adding it as a reviewer in the review tool, and it is not possible for the developers to give feedback to Review Bot regarding the usefulness of the analysis. To investigate the effectiveness of Review Bot, VMware conducted a review where a group of developers provided feedback on 1000 comments generated from Review Bot. Through this process, they found that the developers agreed to fix more than 96 % of the comments, and only had concerns about 14.71 %

of the accepted comments. Unlike Tricorder, which integrated and conducted the experiment in the normal developer workflow, the Review Bot experiment was conducted by asking seven developers to provide feedback regarding the 1000 comments from review bot. These comments were related to code changes made in a previous stage of an ongoing project.

Khasiana is a program analysis system developed by IBM [29]. It was developed to address the usability problems of false positives that come with the use of static program analysis [2]. Khasiana integrates three analysis tools (FindBugs, SAFE, and Xylem). To use the system, developers upload code to an online portal, either manually or through a build tool, where the code will be analyzed. The results from the analyzers can then be viewed directly online, and the uploader can give feedback by clicking on one of the feedback buttons ("Invalid", "WontFix", "Confirmed", and "Not Attended"). To evaluate Khasiana, pilots were run for multiple teams within three different groups at IBM (in total 12 teams). Every team had one or two persons in contact with the Khasiana developers. At the end of the pilots, feedback was collected in the form of a questionnaire answered by the contact persons. Via the questionnaire, the developers of Khasiana found that between 20 % and 75 % of found defects were valid, depending on the team. In addition to the questionnaire, anecdotal feedback indicates that the system was well received, and features such as the ability to tag defects and to give feedback were some of the features of Khasiana that were appreciated by developers.

MEAN is a data-driven meta analysis system, built on similar principles as Tricorder [5], Tricium [6], and Shipshape [30]. It is made up of several micro-services [7]. These services communicate via messages, and when a code change is detected, a message is sent to MEAN detailing which files are changed, so that the appropriate analyses can be started [8]. MEAN integrates several analyzers, such as Pylint, ShellCheck, and Hadolint, but the system can be extended with additional analyzers. The results of the analyses are sent as messages, which can be received by several different services. These messages are also sent directly to the review system, as in-line robot comments, with the option for developers to mark comments as 'NOT USEFUL'. The developers of MEAN deployed the system at Axis for seven weeks [7]. Via user feedback, the MEAN developers found that the main negative user experience with MEAN was that robot comments were published for whole files. In reality, a lot of the results were unrelated to the most recent code changes, causing a lot of noise for the developers.

Chapter 5
The MEAN System

MEAN is a meta analysis tool which automatically runs relevant program analysis at code changes. The analysis results are then presented for the developer as robot comments. At the same time, feedback is gathered from the developer via a 'NOT USEFUL' button regarding the usefulness of the analyses. The structure of MEAN is that of several micro-services, which all perform a specific task. A generic overview of the MEAN system and its integration points can be seen in Figure 5.1. These micro-services communicate with each other via four different kinds of messages. For example, it is the job of the `MEAN Publisher` service to detect code changes from the review system, and send out a `MEAN-Request` message. This message is received by the `Main MEAN System`, informing it that a code change has occurred, and that some analyses should be executed. Most of the services are written in the programming language Go, but the `MEAN Publisher` is for example written in Python. One advantage of a micro-service design is that if one service does not fit a certain tool chain, then it can relatively easy be rewritten, without having to modify any other service.

In the following sections we describe the different services, message protocols, and configurations of MEAN. First we describes how the different modules communicate. Then we describe the different MEAN modules by following the life of a change event. Where it starts with a code push to the review system, Gerrit, ending with robot comments with analysis results being published back to Gerrit.

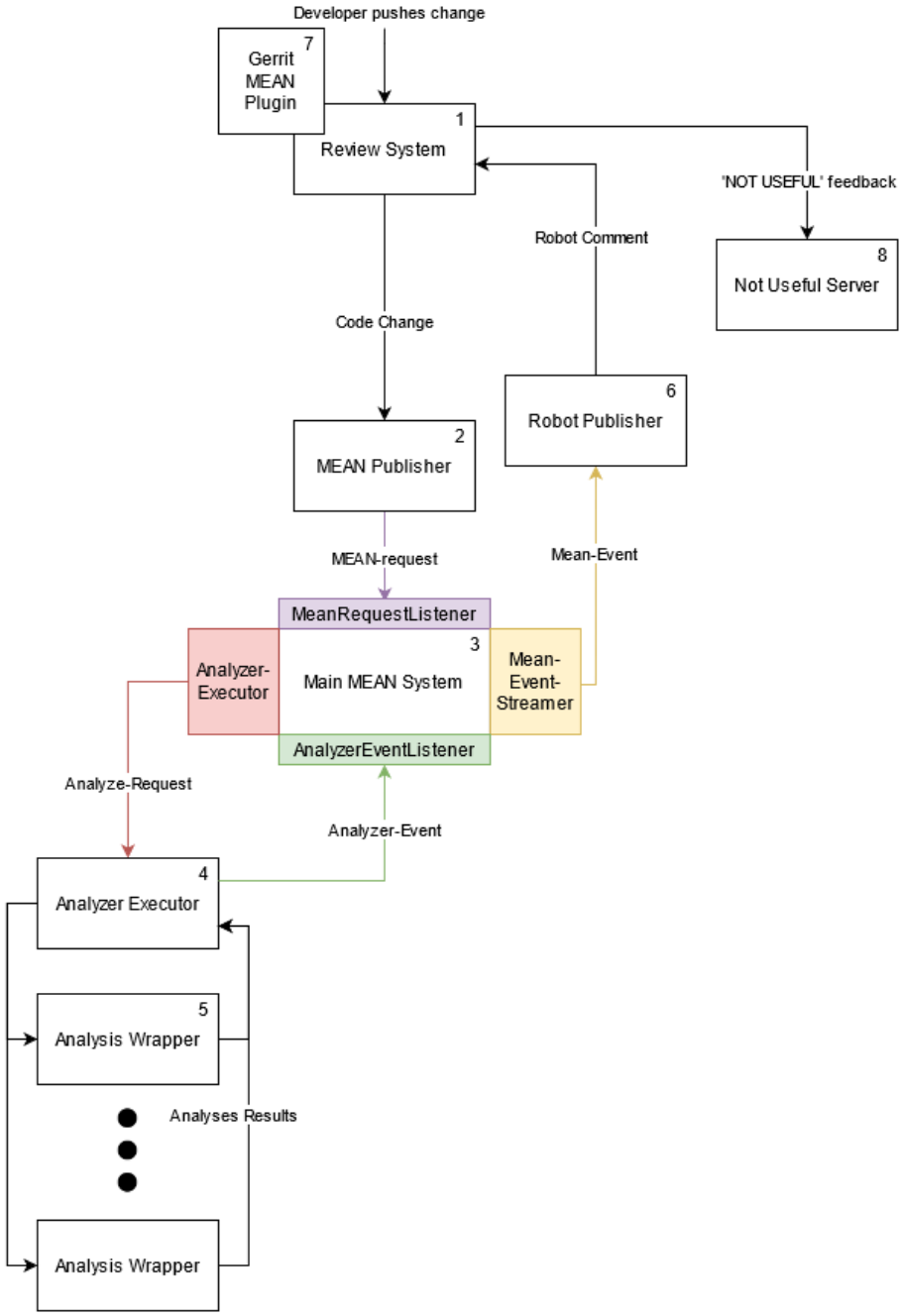

Figure 5.1: Overview of the MEAN system.

5.1 Message Protocols

RabbitMQ is the message broker that is used to transfer messages between the different MEAN services [24]. RabbitMQ is made up of producers, consumers, queues and exchanges. Producers create and send messages together with a routing key to an exchange, and then the routing key determines which of the queues connected to the exchange that the message should be forwarded to. A consumer receives messages from a specific queue. This means that in MEAN, the `MEAN Publisher` service is a producer that produces `MEAN-Request`

messages, which are consumed by the `Main MEAN System`. RabbitMQ is explained in more detail in Section 2.5.

RabbitMQ was chosen as message broker for MEAN because it was already part of the tool chain at Axis where the original MEAN master's thesis were integrating it into [7]. However, the system can of course be tweaked to use another message broker system.

Four different kinds of messages are sent between the MEAN services. These messages are formatted using JSON. JSON is a stateless human-readable data format. The four types of messages are detailed below:

- `Analyze-Request` (red arrow): Sent by the `Main MEAN System` to the `Analyzer Executor`, and contains information about what analysis to run, and what source files to analyze.
- `Analyzer-Event` (green arrow): Sent by the `Analyzer Executor` to the `Main MEAN System`, and contains information about the result of the analysis.
- `MEAN-Event` (yellow arrow): Sent by the `Main MEAN System` when an event occurs. This message type encapsulates another message. For example, when the `Main MEAN System` receives an `Analyzer-Event`, it will send out a `MEAN-Event` encapsulating that `Analyzer-Event`.
- `MEAN-Request` (purple arrow): Sent by the `MEAN Publisher` to the `Main MEAN System` to inform about a code change, and contains information about which source code files were changed.

5.2 MEAN-publisher

The `MEAN Publisher` (2 in Figure 5.1) is the first MEAN module to be run when a change event occurs in Gerrit. It is started in a Docker container via a Jenkins job, which in turn is triggered by a change event in Gerrit, detected by the Gerrit Trigger Plugin. The task of the `MEAN Publisher` micro-service is to inform the MEAN system that a code change has occurred, so that appropriate analyses can be run. This is done by sending a `MEAN-Request` message to the `Main MEAN System` via a RabbitMQ exchange. With the message containing information about which source files were changed and what the source context is (for example which project the changed files belong to).

5.3 Main MEAN System

When the `MEAN Publisher` sends out a `MEAN-Request`, it is received by the `Main MEAN System` (3 in Figure 5.1). Via the `MEAN-Request` and some configuration, the `Main MEAN System` calculates what analyses are appropriate to run, and sends out `Analyze-Request`s based on that information. Other that sending `Analyze-Request`s, the `Main MEAN System` is the core of MEAN, and has a lot of other responsibilities. It binds together the other systems, and is composed of five interfaces and a configuration:

- `AnalyzerStates`: The analyzer states interface keeps track of the current state of currently executing analyzers.

- `AnalyzerExecutor`: The analyzer executor interface sends `Analyze-Request` messages to inform the `Analyzer Executor` that some analysis should be done.
- `MeanEventStreamer`: The MEAN event streamer interface reports MEAN events from the `Main MEAN System` to other services.
- `AnalyzerEventListener`: The analyzer event listener interface handles incoming analyzer events.
- `MeanRequestListener`: The MEAN request listener interface handles incoming MEAN requests.
- Configuration: The `Main MEAN System` holds the configuration for MEAN. The configuration determines for example how long an analyzer is allowed to run before timeout, and which, if any, analyzers are blacklisted and should not be run.

This means that an implementation of the `Main MEAN System` must implement these interfaces and the configuration. The `Main MEAN System` is implemented by two different services in MEAN, `localanalyze.go`[1] and `jenkinsanalyze.go`[2]. The local analyze service is a very simple implementation of the `Main MEAN System` which does everything locally on the machine. It takes command line arguments to determine what files to do analysis on, sends `Analyze-Request`s to a local `Analyzer Executor`, and then prints the resulting `Analyzer-Event`s to the console.

The Jenkins analyze service is more complex, and uses RabbitMQ to communicate. For example, it sets up a MeanRequestListener that listens for `MEAN-Request` messages on a specified RabbitMQ queue.

5.4 Analyzer Executor

When the `Analyzer Executor` (4 in Figure 5.1) micro-service receives an `Analyze-Request` message from the `Main MEAN System`, it runs the appropriate analysis, by starting an analyzer wrapper. This is implemented via a Jenkinsfile with five different stages: `create input files`, `checkout`, `run analyzer`, `publish`, and `cleanup`. First, in `create input files`, a well-defined directory structure is created, which the analyzer will have access to. An overview of the directory structure can be seen in Figure 5.2. Then, the `Analyze-Request` that the service received from the `Main MEAN System` will be written to a file in the input directory. An `Analyzer-Event` message will then be sent, informing the `Main MEAN System` that the analysis has started. In the next stage, `checkout`, the `Analyzer Executor` pulls down the code to be analyzed from the relevant Git repository, into the code directory. After that, in the `run analyzer` stage, the actual analysis is run, and the result is sent as an `Analyzer-Event` in the `publish` stage. Last, the `cleanup` stage removes the directory structure.

This micro-service is stack-dependent because it pulls the code to be analyzed from Git, meaning that is has to be re-written if another code storage system is used. However, it is not dependent on RabbitMQ, because it sends `Analyzer-Event`s indirectly via a supplied Docker image, which can be implemented to use any type of message sending mechanism.

[1] Link: https://gitlab.com/lund-university/mean/-/blob/master/cmd/localanalyze/localanalyze.go
[2] Link: https://gitlab.com/lund-university/mean/-/blob/master/cmd/jenkinsanalyze/jenkinsanalyze.go

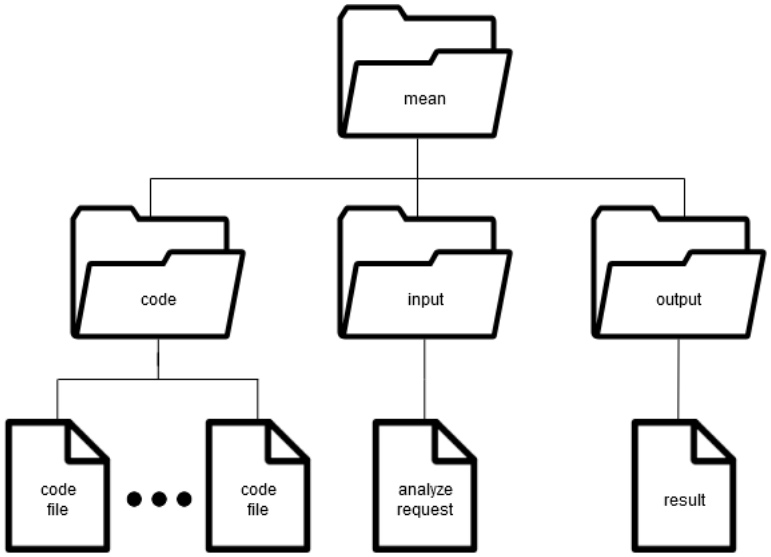

Figure 5.2: The directory structure created by the `Analyzer Executor`, which will be available to the analyzers.

5.5 Analyzer Wrapper

The analyzer wrapper (5 in Figure 5.1) service is a wrapper around a static program analysis tool. It is started in a Docker container by the `Analyzer Executor` when it receives an `Analyze-Request` message. It is the task of this service to use the input provided by the `Analyzer Executor` (the analyze request and code files in Figure 5.2) and start the actual analysis on the relevant files. It is therefore also the responsibility of the Analyzer Wrapper to not run the blacklisted analysis categories as specified in the Analyze Request. After the analysis has run, this service transforms the output from the analysis into a well-defined JSON format. This format contains a list of code violations (called notes) that the analysis found, and a list of errors, which it saves in the result file (as shown in Figure 5.2). This result file is then included in the `Analyzer-Event` message sent by the `Analyzer Executor` to the `Main MEAN System`.

MEAN comes with a number of already written Analyzer wrappers for some well-known analyzers, such as Hadolint, Pylint, and Shellcheck.

5.6 Robot Publisher

When the `Main MEAN System` receives an `Analyzer-Event` from the `Analyzer Executor`, it wraps the `Analyzer-Event` in a `MEAN-Event` message and sends it out via RabbitMQ again. These `MEAN-Event` messages are then picked up by another module, namely the `Robot Publisher`. The `Robot Publisher` (6 in Figure 5.1) micro-service, written in Go, handles the communication with the code review tool, in this case Gerrit. When it receives a `MEAN-Event`s on a RabbitMQ queue, it checks if it contains an `Analyzer-Event` of the type result. If it does, it will convert the event into a robot comment, which it sends to Gerrit via HTTP. In addition to this, the service also sends the robot comment via RabbitMQ, in case other services need it. Also, before sending the robot comment, it queries Gerrit via a HTTP GET

request, to find out if MEAN has been disabled for that particular project. If MEAN has been disabled for that particular Gerrit project, the robot comment is instead only sent via RabbitMQ with a special routing key. This is so that other services, if needed, can know what comments were not published.

5.7 Gerrit MEAN plugin

When Gerrit receives a robot comment from the `Robot Publisher`, it associates the comment with the correct code change and file. However, some additional functionality is needed. This functionality is implemented by the Gerrit MEAN plugin (7 in Figure 5.1). The Gerrit MEAN plugin is not strictly speaking a MEAN micro-service, but rather a plugin inserted directly into Gerrit. The job of this plugin is to add the 'NOT USEFUL' button to MEAN robot comments, giving the developer the ability to give feedback to the MEAN system regarding the robot comments.

When a developer clicks on a 'NOT USEFUL' button, a HTTP POST request is sent to a specified host with information about the clicked comment. The plugin also adds the ability to configure MEAN via a configuration file in the Gerrit repository. In this file, it can be specified whether or not MEAN is enabled for the project, the host computer to send the 'NOT USEFUL' feedback to, and individual configuration for every analyzer, such as the timeout and blacklist. With this configuration comes three representational state transfer (REST) endpoints, `/meanhost`, `/meanenabled`, and `/meananalyzers`, which can be queried to find out the configuration for a certain Gerrit project. For instance, the `Robot Publisher` service queries the `/meanenabled` endpoint to make sure that the project has set MEAN to enabled before sending robot comments.

5.8 Not Useful Server

When a developer clicks the 'NOT USEFUL' button on a robot comment in Gerrit, the `Not Useful Server` (8 in Figure 5.1) picks them up. The last of the MEAN modules, the `Not Useful Server` is implemented as a small python script using the flask framework. When running, this service listens to HTTP POST requests from the Gerrit plugin. When such a request is received, it means that a developer has clicked the 'NOT USEFUL' button on a robot comment in Gerrit. Upon receiving a HTTP request, the service extracts the data from the request, and forwards that data to an exchange via RabbitMQ. This means that another service can listen to the RabbitMQ exchange via a queue and appropriately handle the 'NOT USEFUL' messages. For example, storing them in a database, or even dynamically disable certain analysis categories based on 'NOT USEFUL' messages.

Chapter 6
Integration of MEAN at Bosch

In this chapter, we detail the steps we took to integrate MEAN into the Bosch tool stack. Figure 6.1 shows an overview of the system components when it was integrated into Bosch.

6.1 Bosch Tool Stack

The production part of the Austrian team mainly used two in-house analyzers. The first, called the Framework Analyzer was organized by having a large framework, written in Java. This framework made it easy for any developer to create a new program analysis (called 'plugin') which could then be used via the framework, either locally or in Jenkins. A large number of plugins were already present (about 50), but unfortunately, the output from said plugins were not standardized. Instead, the output was written out as free text, one line for every defect message.

The second tool that we decided to wrap was also written in Java, called the Correct Mappings Analyzer. This tool was unrelated to the framework system that is used for the Framework Analyzer. The purpose of this analyzer was to make sure that the mappings of certain dependencies followed a predetermined standard. It also checked that a certain requirement was properly tagged. This tool only produced one result per build related to the whole repository. The output of the two above mentioned tools was typically printed to a large Jenkins log.

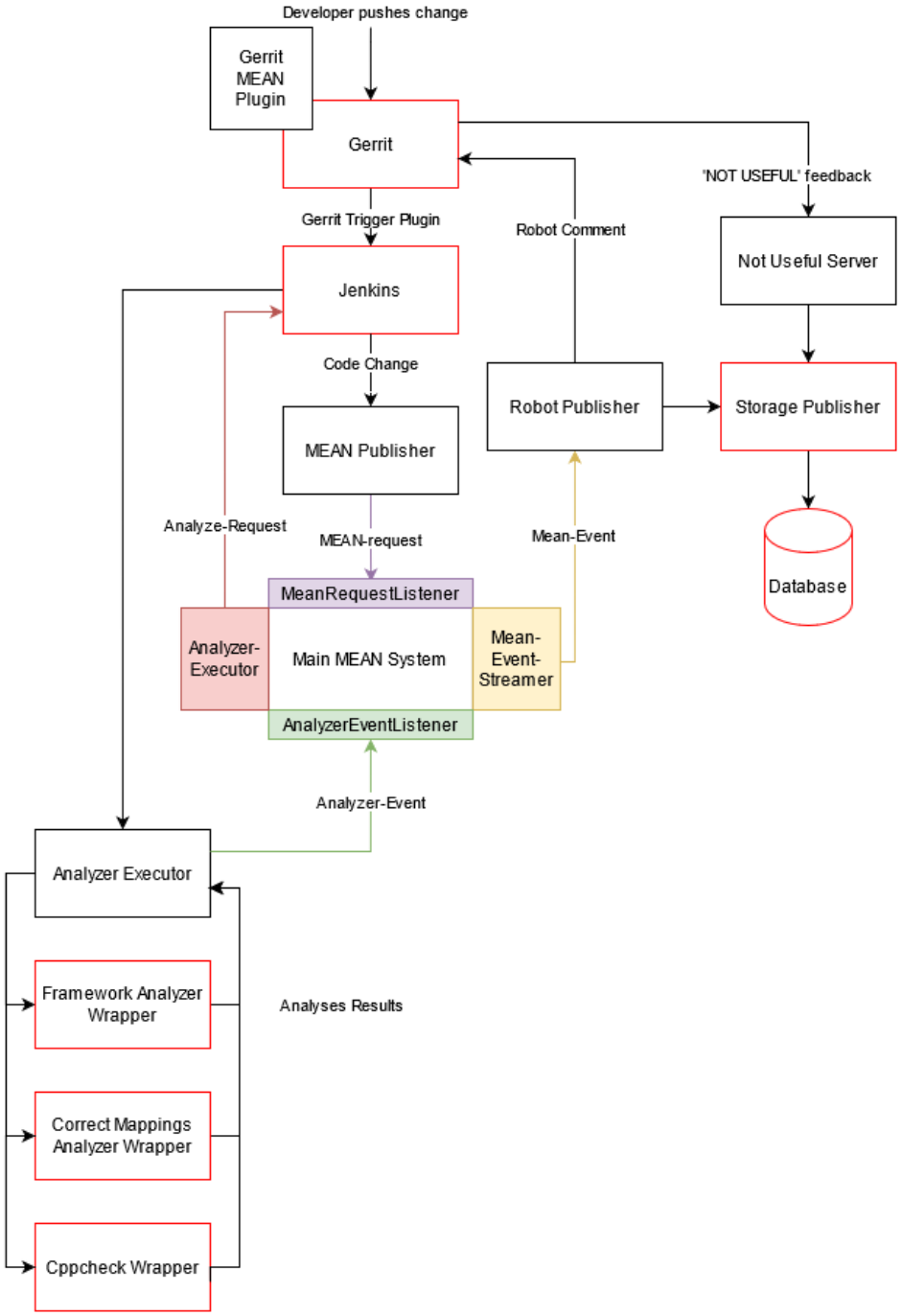

Figure 6.1: Overview of the MEAN system integrated into Bosch. The added and changed pieces are outlined in red.

6.2 Pre-Integration

As a first step, before integration into the Bosch tool stack, we set up MEAN locally on a computer. This also involved setting up and configuring local variants of the systems that MEAN communicate with, i.e. Gerrit, Jenkins, and RabbitMQ. In addition to this, we also installed and configured the necessary Jenkins and Gerrit plugins. Along the way, we created

a README file, documenting the different steps that need to be taken to get the system running. We did this to make the integration easier in the future. Also, we created a few scripts such as a small bash script that automatically sets up and configures the needed RabbitMQ exchange, queues and routing keys. The reason for setting up MEAN locally before integration into Bosch, was that the Bosch integration would hopefully go smoother with prior integration experience. Also, we could use the local setup as a first testing stage for new MEAN improvements and analyzer wrappers. This would then allow us to make sure that everything was working correctly before deploying the changes to Bosch.

While setting up MEAN locally, we discovered a few minor bugs in the system, such as erroneous output format and parsing negative numbers from JSON files into unsigned integer types. We documented these bugs, and after the local setup was completed, fixed the bugs and pushed the fixes to the open source MEAN Git repository.

Besides the README file, we kept some notes about the integration areas where there would be potential issues, so called integration problems (**IP**s):

- **IP$_1$**: Some kind of central repository for Docker images would be needed, so that the `Analyzer Executor` module would be able to fetch and run appropriate analyzer wrapper images.
- **IP$_2$**: As for the Gerrit plugin, one potential problem was that the plugin is hard-coded to only accept robot comments for a user named 'svcmean', which would mean that such a user would either need to be created, or the plugin code would have to be changed and re-built.
- **IP$_3$**: Another potential hardship is RabbitMQ. If Bosch is not already using RabbitMQ as a message broker, then it would need to be set up and configured, ideally with a public key infrastructure so that Transport Layer Security (TLS) could be used for encryption and authentication.

6.2.1 Test Integration

As an intermediate step between the local MEAN setup and the actual integration of MEAN into the production Gerrit, we set up a Gerrit test instance as the integration point. The idea was that by first integrating MEAN into the test Gerrit, we could work out the potential problems that would arise when integrating into Bosch without interfering with the production systems. After that the switch to the production Gerrit would be smoother. It was in this stage we solved most of the problematic integration points mention in Section 6.2.

As for **IP$_1$** (finding a central Docker repository) mentioned in Section 6.2, the `Analyzer Executor` would always run on the same machine. Therefore, we decided to not use a central Docker repository for the analysis wrapper images, but instead keep the images locally on the machine. While there exists a central Docker repository at Bosch, there would still be some hassle with getting access to it and upload the images. Thus, we decided that keeping the images locally on the `Analyzer Executor` machine was the easiest and fastest solution.

IP$_3$ (set up RabbitMQ) was also resolved in this part of the integration. RabbitMQ was not used previously by Bosch, so we set it up and configured it in this stage. After some investigation, we skipped TLS, because time was running short. It would also be too large of a task to set up the whole public key infrastructure that was needed. Since everything was

running on the Bosch VPN, and no real sensitive data is sent via the MEAN messages, this was a reasonable compromise.

6.3 Integration Stage 1

Unfortunately, when the time came to integrate MEAN into the Bosch tool stack, we discovered that there are very strict rules regarding adding new Gerrit plugins at Bosch. This was an issue, because the rest of the MEAN system cannot work without the Gerrit plugin. This is because the plugin is the component that the rest of the system queries to get the current configuration. A large part of the MEAN modules would have to be rewritten, and the 'NOT USEFUL' feedback button would be gone. Without the button, developers would not be able to give feedback about the analysis results. Fortunately, no rewrites were needed, since we identified another Bosch team, which had larger control over their Gerrit server. Therefore, we made the decision to integrate MEAN into their tool stack. However, this setback led to a delay in the deployment of MEAN.

6.3.1 Production Integration

After we had fully integrated and tested the MEAN system with the Gerrit test instance, we moved it to the production Gerrit. This meant installing the MEAN Gerrit plugin on the production Gerrit, as well as adding the appropriate MEAN configuration. To resolve the problem with the hard-coded Gerrit username, $\mathbf{IP}_2$, we changed the username of the Gerrit service user to an already existing service user in the production Gerrit. In addition to that, we reconfigured the Jenkins scripts to trigger on the production Gerrit instead of the test Gerrit. As a final small step before enabling MEAN for the developers, we changed the `Robot Publisher` to only publish comments to a special test repository on the production Gerrit. This would help us to confirm that the switch to production was fully working before enabling MEAN for everyone.

We discovered in this stage that the 'NOT USEFUL' button was not working, due to the plugin complaining about a missing function. It turns out that MEAN was never tested for the Gerrit version that the production system used. The plugin used a function `getAccount()` to get the username of the current user to send with the 'NOT USEFUL' feedback. This function was however not added until a later version of Gerrit. We solved this by replacing the function with another function `get("/account/self")` which does the same thing.

Another problem that arose was that there were some incompatibilities with Linux and the Framework Analyzer that was wrapped into MEAN. This was a problem because the machine which were assigned to us that was running the `Analyzer Executor` module was a Linux computer. In the end, we solved this issue by doing a few small modifications to the source code of the Framework Analyzer. We also notified the maintainer of the Framework Analyzer to make them aware of the incompatibilities.

Chapter 7
Mean Extensions

Before integration into the Bosch tool stack, we needed to complement MEAN with a storage module addition to collect quantitative data, as can be seen in Figure 6.1. Therefore we implemented a `Storage Publisher` micro-service that stores analysis results and 'NOT USEFUL' feedback in a database. In addition to this, Figure 6.1 shows the analyzer wrappers that we implemented to produce robot comments in Gerrit. We also implemented two new modules, a `Filter Module` and a `Visual Feedback Feature`, to find an answer to $\mathbf{RQ_1}$ by reducing analysis result noise.

7.1 Storage Publisher

The `Storage Publisher` is a micro-service written in Go, with the task of publishing data to a database. Such a module did not already exist in MEAN, because Axis, where MEAN was originally integrated, already had a solution set up for automatically storing messages sent on RabbitMQ [7]. Thus, we found a need to create such a module, so that we would be able to gather data about analysis results and 'NOT USEFUL' feedback.

It works by connecting to a MongoDB database. MongoDB works well as a database for storing MEAN messages, since it is a document-oriented database. As such, the messages can be immediately stored in JSON format, without first translating it to a table format. This would otherwise be needed for storage in a relational database. The `Storage Publisher` listens to a RabbitMQ queue, where it receives messages from the `Robot Publisher` and the `Not Useful Server` services. When it receives a message, it enters it into the MongoDB database. In the end, we uploaded this module to the MEAN open source repository, where we improved upon it through several iterations based on feedback from reviewers. Finally, being approved and then merged into the repository.

7.2 Analyzer Wrappers

Before we integrated MEAN into Bosch, we wrote a few new analyzer wrappers.

7.2.1 Implementation

We implemented the analyzer wrapper for the Framework Analyzer as a single wrapper, written in Python and contained in one Docker container. This wrapper is responsible for reading the analysis request message and then starting the framework with the selected plugins. In addition to the wrapper, we implemented several parsers in python, one for each plugin that the wrapper starts. These parsers are supplied to the wrapper, which uses the appropriate parser to parse the output from the plugins into the JSON format that MEAN requires. We used the plugin names as the category attribute in the analysis results, so that different plugins could easily be enabled and disabled via the blacklist in the MEAN configuration. This implementation also allows us to add with ease another framework plugin to MEAN, by writing a new parser, supplying it to the wrapper, and updating the Docker image. In the end, we created parsers for six plugins, after having consulted with the maintainer of the Framework Analyzer. This helped us find out which plugins were most appropriate to integration into MEAN, taking criteria such as noisiness and usefulness for developers into account.

Similar to the above implementation of the Framework Analyzer, we wrapped the Correct Mappings Analyzer with a python script placed within a Docker container. Since the analyzer produced one result for the entire project, no file existed for the robot comment to be attached to. We therefore chose to attach the analysis result comment to the commit message.

7.2.2 Alternatives Considered

Below, we detail other implementation approaches we considered for the Framework Analyzer. Figure 7.1 shows a UML representation of the different approaches.

- Ad Hoc Wrappers: One possible solution would be to have one isolated wrapper for every plugin, similar to how the current MEAN wrappers are designed. However, this is only an acceptable solution when wrapping a small number of plugins, since unnecessary code duplication would otherwise become a large problem. This approach would also mean one Docker image for every wrapper, which comes with both upsides and downsides. One upside would be that the `Analyzer Executor` could potentially start several wrappers in parallel, which could speed up execution. However, one downside would be the extra hassle of keeping track and maintaining a large number of Docker images, as well as have near-identical Dockerfiles for every wrapper.

- Object-oriented Wrappers: An improvement of the previous approach would be to create an abstract super wrapper, which already implements most of the functionality that is identical for the different plugins, such as starting the Framework Analyzer and reading the analysis request. Then, a wrapper for every plugin would be created that inherits from the super wrapper, so that only the parts specific to the plugin (e.g. the output parsing) would have to be rewritten for each wrapper. As with the above

approach, this would mean one wrapper and image for every plugin, but without the problem of code duplication. However, unlike the chosen approach, this would still result in a large amount of near-identical Dockerfiles.

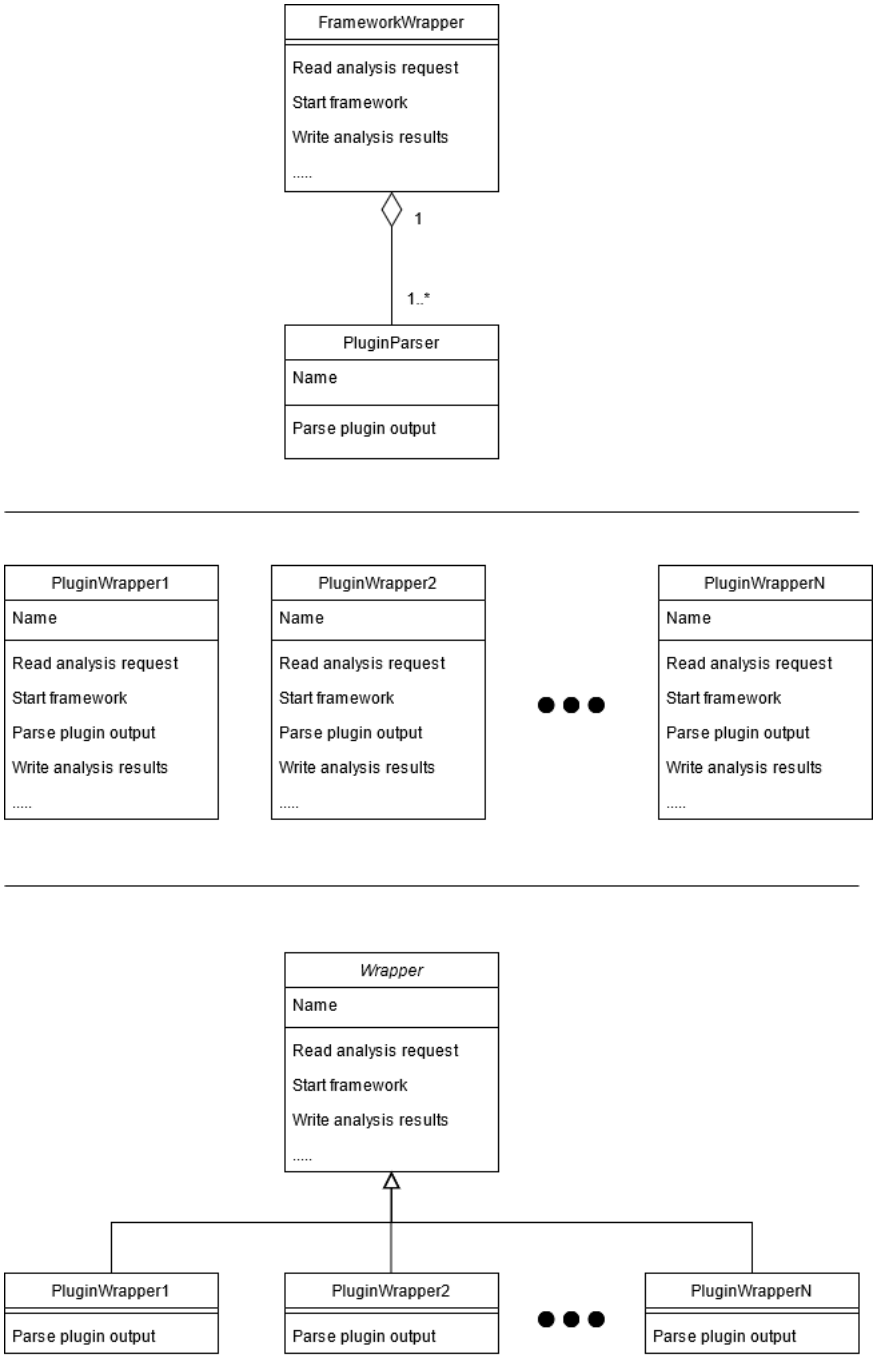

Figure 7.1: UML diagram for the three different implementation alternatives for the analyzer wrappers. From top to bottom: Chosen approach, Ad Hoc Wrappers, Object-oriented Wrappers.

7.3 Filter Module

The purpose of the `Filter Module` is to filter out results that are not connected to the changed lines of the most recent code change.

7.3.1 Background

Currently, the MEAN system always runs analysis on whole files, regardless of how small the most recent code change in that file actually was. This means that developers might get bombarded with analysis results that are not relevant to the specific code change that the developer wrote. In fact, a large part of the analysis result might pertain to code written by another developer entirely, which the current developer might not have enough knowledge about to fix. The advantage of filtering such results would be that the developers can then focus on the results relevant to their code change, without having to sieve through less relevant results.

7.3.2 Implementation

We implemented the filtering as a separate micro-service written in python, to reduce the amount of changes needed to the rest of the system (see Figure 7.2). Since the Jenkins machine might not have python installed, we opted for the service to be placed in a Docker container. The container would then be started by the `Analyzer Executor`, after the analysis was done running. Our reason for inserting the filtering service at this specific part of the MEAN system is that the service will then have access to the Git repository of the project that was analyzed. With this, it could then use, for example 'git diff' to determine which lines belonges to the most recent change. Running a filter service would of course incur some performance cost, however nothing too dramatic. The `Analyzer Executor` already starts a number of containers (for the analysis itself, and every time it sends an analyzer event message). Therefore, starting a container for the filter would have minimal impact. The filter does not increase network traffic either, since it works on the local Git repository, performing the filtering by manipulating the output file from the analysis, shown in Figure 5.2. In fact, the filter will reduce the network communication, since it will filter out analysis results, thus making the final analysis result message smaller. The performance cost of the filtering itself is unlikely to be a problem. According to the diff manual, 'git diff' uses the Myers algorithm together with a heuristic, and has a time complexity of $O(N^{1.5}logN)$, where N is the sum of length of the two files that are compared [31] [32]. To keep the changes to MEAN as small as possible, we let the state management of the `Analyzer Executor` remain unchanged. This meant that, to the `Main MEAN System`, the analysis is still in the 'started' state while the filter is running.

Algorithm 1 describes the steps taken in the filtering module to achieve what it is supposed to.

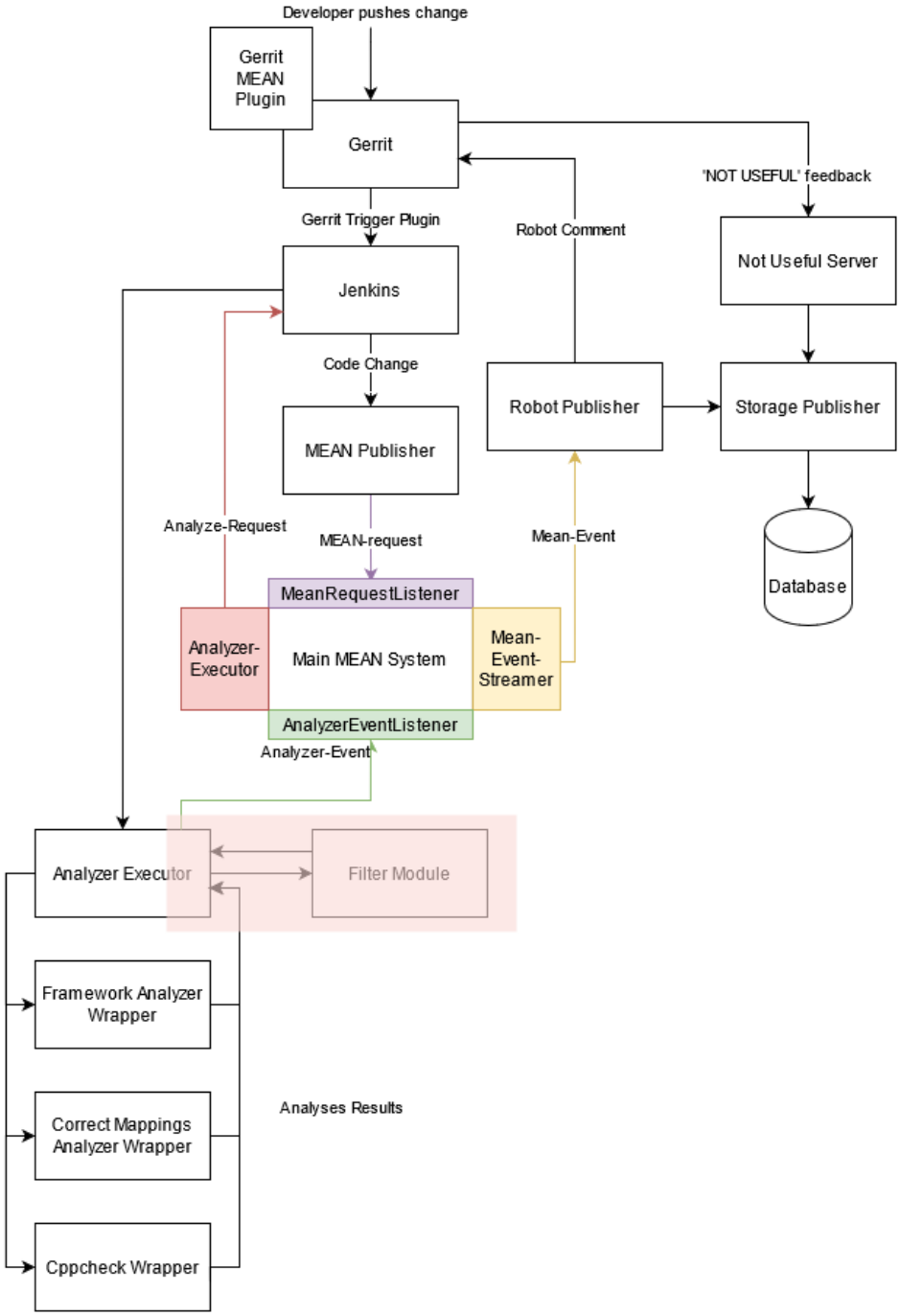

Figure 7.2: The overview figure of the MEAN system, with the `Filter Module` added, highlighted in red.

7.3.3 Alternatives Considered

Other approaches for filtering are possible, some more advanced. Here is a list of approaches we considered:

- Run the analysis before and after change: By comparing analysis results from before and after a change, it could be determined with high precision which results are relevant to

Algorithm 1 Overview of steps taken in module to filter analysis results.

```
Run git diff
Use git diff output to determine changed lines for analyzed files
Parse analysis result file '/mean/output/result.json'
Loop through notes in result and filter out based on git diff
Save filtered result again to file '/mean/output/result.json'
```

the most recent change. The problem with this approach is that it is more complicated, and requires larger changes to MEAN. Another difficulty with this approach is how to compare the analysis results from before and after. To obtain maximum precision, not only would the result description and category need to be compared, but also the location of the result, which is difficult since the line numbers will have changed when the file was changed.

- Let analyzers handle filtering: The responsibility of filtering could be on the analyzer wrappers themselves. However, this would mean that the existing analyzers would have to be rewritten, and there would likely be code duplication between the analyzers. The advantage of this approach is that the filtering can be more tailored to the specific analysis. For example, it could be easier to determine that 'Unused Import' defects are connected to the most recent change, even though it is not near the changed lines.

- Gerrit filtering: By letting the Gerrit plugin handle the filtering, other filtering criteria could be used, for example filtering based on if the same results is still unresolved from an older code change. Another advantage with this approach is that it could be possible for the user toggle between showing/hiding the filtered results. The disadvantage of this approach is that it would require large changes to the Gerrit plugin.

7.4 Visual Feedback Module

Our purpose for adding this module was to add functionality, such that noise is reduced within a review by visually reacting to 'NOT USEFUL' clicks. A background is given describing the motivation for the addition of the feature, followed by its implementation and concluding with alternative approaches we considered.

7.4.1 Background

The 'NOT USEFUL' button is meant to signal to the maintainers of the tool, or the MEAN service itself, that the specific analysis result is not satisfactory. This type of result can range from it being a false positive to unsatisfactory guidelines. But even though a developer marks the result as 'NOT USEFUL', it will still remain as a robot-comment in the current and future patch sets of the review context. Hence these "sticky" 'NOT USEFUL' comments produce noise to the developer and reviewers as superfluous information. Thus, noisy tools can reduce the meaningful time the developer and reviewers spend on work. Having to deal with the "sticky" 'NOT USEFUL' comments will thus cost effective time that would otherwise benefit the development if spent on meaningful issues.

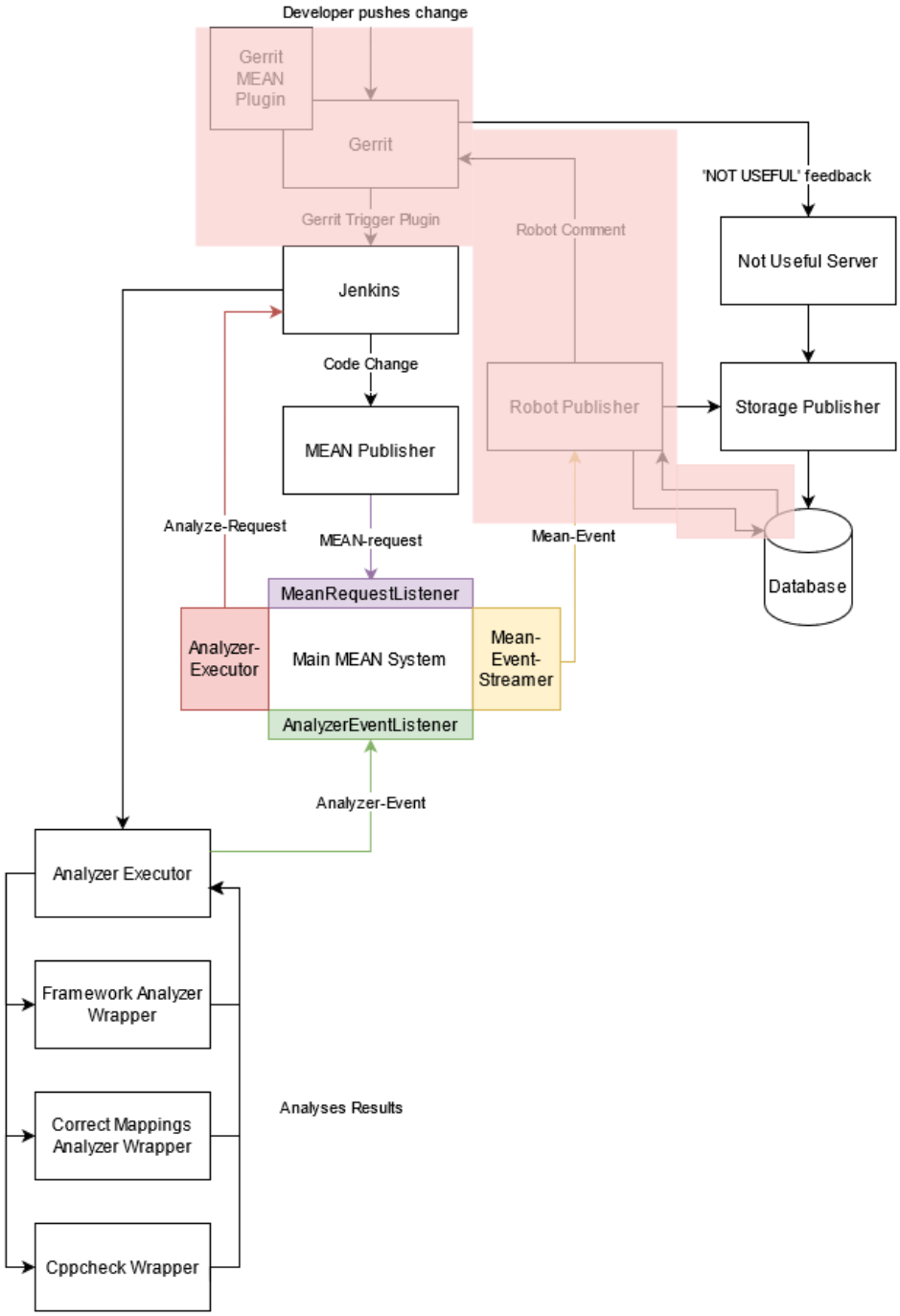

Figure 7.3: An overview detailing the scope, highlighted in a light tone of red, where changes were made to implement the `Visual Feedback Feature`.

7.4.2 Implementation

Due to time and workload constraints placed by the scope of this thesis, we decided to only implement the core functionality. Namely, the mitigating of the above described issue developers are facing with robot comments. One of our main reasons for this decision was due to the complexity of tracing analysis results that are caused by the same piece of code.

Another issue that arises when taking one of the approaches we detail in Section 7.4.3, is configuring the changes done to the robot comment to-be proportional to the multiple users' actions. The topics we describe in Section 7.4.3 can then be used as starting points for future work for the core functionality described below.

Thus, we implemented the functionality such that it responds to the 'NOT USEFUL' button being pressed. When this happens, the color of the robot comment is changed to signal to all involved that it has been pressed. This change will remain for the rest of the review process, starting from next patch set, if two robot comments are equal. Robot comments will only be seen as equal if the code that caused it is identical in the previous patch set. That is, it has not been modified, had its line number changed or placed in another file.

To implement this feature, we needed to make two changes, one in the Gerrit MEAN plugin, such that it can modify the color of the robot comment. The other change we made was to the `Robot Publisher`, such that it can find out if a to-be published comment has had its 'NOT USEFUL' button pressed in a previous patch set. The scope of the changes are highlighted in Figure 7.3. We chose the `Robot Publisher` since all the robot comments pass through this module. This would then help us minimize coupling in the MEAN system compared to other entry points. To signal that a robot comment was 'NOT USEFUL', we added an additional field to the robot comment inputs properties. These robot comment inputs, encapsulate an analysis result, one for each result sent to Gerrit, where it is then used to create a robot comment.

Thus, we attached a conditional in the attached function within the *mean-feedback-button.js* file from the Gerrit MEAN plugin. It checked whether the value of the "not_useful_pressed" in the properties field had been set to "true". If so, the color of the robot comment was changed to gray.

When the 'NOT USEFUL' button is pressed, a message to RabbitMQ is sent, which is then stored in a mongoDB database collection. Two collections, not_useful and published_notes, in this database are used to check whether a robot comment has had its 'NOT USEFUL' button pressed in a previous patch set. The entries in the not_useful collection contain data as detailed in Listing 7.1. Since these do not contain enough information to test for equality between robot comments, we combine the information with the published_notes collection, containing the robot comment information as seen in Listing 7.2.

Listing 7.1: note_useful

```
{
  "note_id": "<value>",
  "analyzer_name": "<value>",
  "category": "<value>",
  "account_info": {
    "username": "<value>",
    "email": "<value>"
  }
}
```

Listing 7.2: published_notes

```
{
  "request_id": "<value>",
```

```
  "analyzer_name": "<value>",
  "note": {
    "category": "<value>",
    "note_id": "<value>",
    "location": {
        "range": {
           "start_line": <value>,
           "end_line": <value>,
           "start_column": <value>,
           "end_column": <value>
        },
        "path": "<value>"
    },
    "description": "<value>"
  },
  "source_context": {
    "project_name": "<value>",
    "ref": "<value>",
    "revision_id": "<value>",
    "change_id": "<value>",
    "host_uri": "<value>"
   }

}
```

Before the data for a robot comment is compiled in the `Robot Publisher`, a query to the published_notes collection is made. This query finds all entries matching with the fields, "analyzer_name", "path", "category", values in "range" and "project_name", of the to-be published comment. This then asserts the equality conditions we defined earlier. For each entry found, a query to the not_useful collection is made to find an entry with the same unique note_id. If such an entry is found, then the robot comment in a previous patch set has had its 'NOT USEFUL' button pressed and thus the not_useful_pressed field will be set to "true".

7.4.3 Alternatives Considered

Most likely the hardest problem to solve is the complexity that arises from tracing robot comments. The same part of the source code that caused a robot comment to spawn may move in the source file, hence changing the line number it existed at previously. Variable and function names may change without affecting the behaviour and syntax of the part of code that caused the analysis result.

Another problem is how we should handle the situation when new reviewers are invited after a change has been made. The easiest approach is to have the newly invited reviewers inherit the current settings and treat these as guests. Thus they will not affect the changes made by the proposals described below. Due to this, they will be unable to cause any change by pressing the 'NOT USEFUL' button, although, the feedback will still be gathered. Below, we list the proposals we considered to solve this issue. All proposals are contained within the

scope of a review.

Gradual Color Shift

Gradually shift the color of the 'NOT USEFUL' button as more users press it. One way is to increase the opacity of a color such as red to easily distinguish it from the rest. Once all users have pressed the button, the comment may possibly be filtered out in future patch sets. Thus overlapping with our proposal to remove the robot comment once everyone has clicked 'NOT USEFUL', although, with no color shift.

The mechanism that keeps track of which user has pressed the button, such that one user cannot continue to press it to further affect the color shift, can either forget or remember each user between patch sets. Remembering users will most likely increase the probability that a specific robot comment is muted.

Pros

- Democratic, all users need to press it to mute the noise.

Cons

- Cumbersome/time-consuming since all users will need to press the button.

Remove Comment

This alternative proposes that the robot comment be removed in response to the 'NOT USEFUL' button being pressed. There are different scenarios for which criteria may need to be fulfilled for this to take effect. Thus, these are detailed below.

Criteria 1: One press on the button to remove the comment. This can be done in several ways, for instance, only the one pressing the button will have the comment muted. Problems that can arise due to this is added complexity to discussion between developers. Since what they see may now differ, thus adding the risk for confusion. Another approach is that it requires only one involved developer to press the button such that comment is removed for everyone involved. The disadvantage to this is that one developer can decide to remove the comment even though others may find it necessary to keep. Thus taking away from the purpose of code review being done as a team.

Criteria 2: Require all involved to press the button for comment to be removed. This is similar to the "Gradual Color Shift" approach, with the gradual shift of color not present, and instead the button is removed at max "color shift".

Collect and Isolate Comments

Collect the sticky 'NOT USEFUL' comments marked as 'NOT USEFUL' as a separate "chat-box". This will thus isolate all unwanted comments to one comment, therefore avoiding removing the comments. As such, they can still be found and viewed should the need arise, but still reduce the noise to the developer by placing all the comments in one post, limiting space taken. An analogy to this is marking email as junk, thus having them be moved to the "junk" folder.

Expanding this comment will then show all the sticky 'NOT USEFUL' comments. This would most likely only be possible to implement as a change that affects all developers and can therefore be seen as an expanded feature to the two above alternatives. For the color shift proposal this would have the added effect that once the color has been "shifted" by everyone present in the review, the comment would be moved to the "junk" comment collection. In the case of the remove comment alternative, the comment would just be moved, as explained above, instead of being removed.

Chapter 8
Results

In this chapter, we present the qualitative data gathered from the interviews we performed, as well as the quantitative analysis data from the MEAN database.

8.1 Qualitative Interview Data

In this section we present the results found from the conducted interviews. The interview participants will be referred to as P1-P4.

The attitude towards code analysis in general from the participants was positive:

> *"In general I think it's a good idea to have static code analysis, ... it should make the life easier for the developers, and detect bugs which wouldn't be detected otherwise."* (P1)

Participants were generally positive about having analysis results show up in Gerrit:

> *"I think that it is a good idea if you present the issues in the review and not somewhere in a file on Jenkins and nobody looks at it."* (P1)

> *"Basically, what I liked was that I was directly informed about the location, where to look to where the problem is."* (P4)

Although some negatives were mentioned:

> *"I think a person can be pretty overwhelmed when there are a lot of errors."* (P3)

> *"This stage is a little bit late. So, I just wanted to have this information earlier, already in the development stage."* (P4)

Something that came up when discussing the published results in Gerrit from MEAN with P2 and P4 was their general development workflow. Something that might explain the quantitative results we gathered:

> *"Before we start developing, we gives some special developer some time to generally check if the tools is working as intended. Then we specify which tools we use for delivery, and then usually no software analyzer issues pop up during development. Because we cannot afford to push the delivery back because of some tool issues that happens frequently anyway."* (P2)

> *"When you send for review, you are usually finished with the development… This is the requirement for us, before you send something for review, you must verify that you pass the static code checks already."* (P4)

We compared what P4 said with CI in the form of many small changes and found out that it was not very efficient for their workflow:

> *"Initially, we also had these ideas to also make some smaller updates, small pushes to a review and everything. … As reviewer, it's really hard for you to get the full picture of the change. … We decided to push the complete source code when it's finished to review. So the reviewer can also check, is something forgotten, is something to be updated, some process related and so on."* (P4)

Another aspect of low amounts of comments stems from the quick tuning that is performed by the tools developers:

> *"The review is blocked anyway if something goes wrong, for false positives, it's a matter of days to fix them."* (P2)

As for MEAN and the feedback loop via the 'NOT USEFUL' button, the response was also positive:

> *"The benefit of it is of course to give the user the power to give feedback."* (P2)

> *"To not waste time of the developer, to figure out if this [an analysis result] is needed or not needed and so on."* (P3)

An interesting issue came up when talking about the 'NOT USEFUL' button in the robot comments, namely that it was very clear what the 'NOT USEFUL' was for, but confusion arose when it came to the 'PLEASE FIX' button:

> *"The only thing I don't really get is the button 'PLEASE FIX', this is also an issue for other developers. It's pretty clear that 'NOT USEFUL' sends that this comment is not useful, maybe change it in the future, or it's a false positive. But they generally don't know what 'PLEASE FIX' means."* (P2)

> *"My interpretation was 'NOT USEFUL' means whatever is written in this comment is not useful, or it's not right or something like this. So it's not useful for me. My interpretation of 'PLEASE FIX' was that this should be fixed by myself now… It was a little bit confusing, what 'PLEASE FIX' means."* (P4)

When we later asked P4 why they pressed the 'NOT USEFUL' button, some issues with Cppcheck became apparent:

> *"We have to provide some files for testing purposes. Which are generated by some kind of tool, we are not allowed to modify them. The CPPchecker then mentions that something is wrong. But I cannot do something about this, because I am not allowed to modify these files. If we do, the tests will not complete anymore. They are not part of the software we deliver."* (P4)

When asked about the two new features to MEAN, nobody had noticed it:

> *"Not sure if I noticed anything."* (P2)

> *"No, I have not seen it actually."* (P4)

But when the features were explained, the response was generally positive. For the `Visual Feedback Feature`:

> *"I think graying out potentially 'NOT USEFUL' comments is a good thing."* (P1)

> *"Maybe it's good to have this key-visual, and also the gray color always in software development... generally means 'please ignore'."* (P2)

And for the `Filter Module`:

> *"In general, this is a good idea ... sometimes if it's hundreds of comments in one file, then it generally takes much time fix a lot of those."* (P2)

But there were some concerns. For the `Visual Feedback Feature`:

> *"I personally would like that maybe only I [the owner of the code change] can press this button, because then ... I know that it is not useful."* (P4)

And for the `Filter Module`:

> *"I think that the developers should be encouraged to fix even unrelated issues, but it should not be mandatory."* (P1)

When we asked what future there is for the MEAN system at Bosch, we discover that there are some hurdles that would need to be overcome for MEAN to be adopted on a more widespread basis:

> *"I mean, if it's working successfully at our department, why not use it at other departments, so I think that there is definitely a chance that other departments will adapt it as well. ... They might not use Git/Gerrit, and they might use something different, then of course it's not so easy to adapt your system."* (P1)

> *"I think it is a very interesting idea. ... I think our department wastes a lot of time figuring out what is working, what is not working, and your system could help understand what is what. ... To be honest, I am not sure, I think you need to have developed infrastructure to use your system. Because a person have to get Gerrit first so to say. This is a long time to go for a lot of departments I guess."* (P3)

8.2 Quantitative Data Management

We closely monitored the robot comments produced after the deployment of the MEAN system into the production Gerrit at the Austrian team. Of which, special interest in 'NOT USEFUL' clicks was maintained. Due to the low number of results produced, which was due to the size of the team, the usage of tuned in-house analyzers, no configurations were done to the analyzers. As such, no analyzer nor specific categories were disabled compared to the thesis done at Axis, where they used a 5% threshold to guide their decisions [12]. We opted out of such methods due to the above limitations as well as an unfortunate timing with the integration of MEAN at the Austrian team, as they were focusing mainly on other work outside the scope of Gerrit.

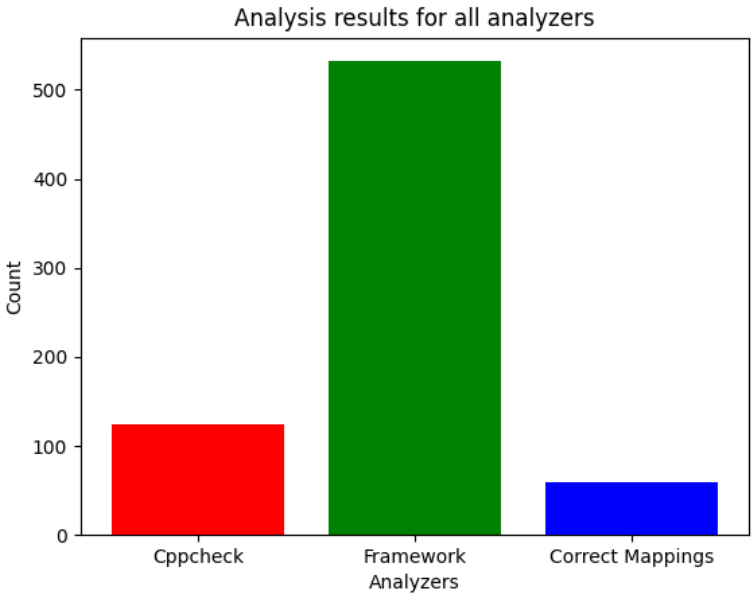

Figure 8.1: The total amount of published analysis results for the different analyzers.

Figure 8.1 shows the total analysis results count for the three different analyzers enabled in MEAN over the whole period of time that MEAN was enabled in the production environment. Cppcheck was enabled 10 days after the other two analyzers, which could account for it having a fairly low count. The Framework Analyzer has some checks that sometimes give rise to tens or even hundreds of results per file, which is why it has such a large count. The Correct Mappings Analyzer on the other hand only gives rise to one (or zero) result for the whole repository, explaining the low count.

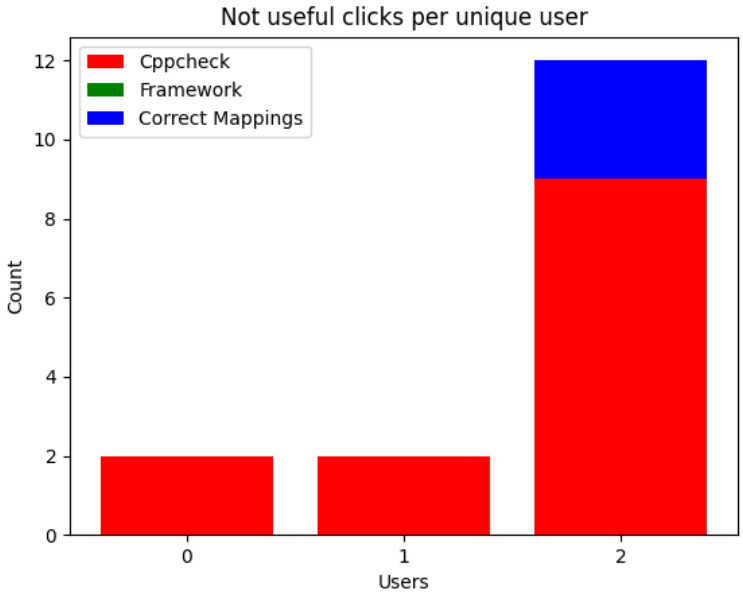

Figure 8.2: The total amount of 'NOT USEFUL' clicks for unique users.

Figure 8.2 shows the 'NOT USEFUL' clicks for each unique user for the different analyzers. As can be seen, only three unique users clicked on the 'NOT USEFUL' button. The names of the users have been changed to a simple id (0-2) for anonymity.

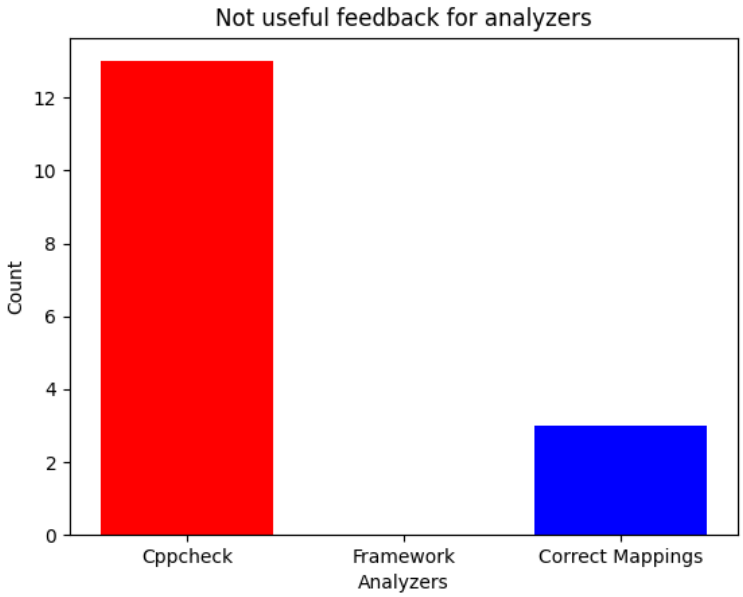

Figure 8.3: The total amount of 'NOT USEFUL' feedback for the different analyzers.

Figure 8.3 shows the 'NOT USEFUL' feedback count for the three different analyzers.

Cppcheck has the largest count of 'NOT USEFUL' clicks, while the Framework Analyzer garnered zero 'NOT USEFUL' clicks.

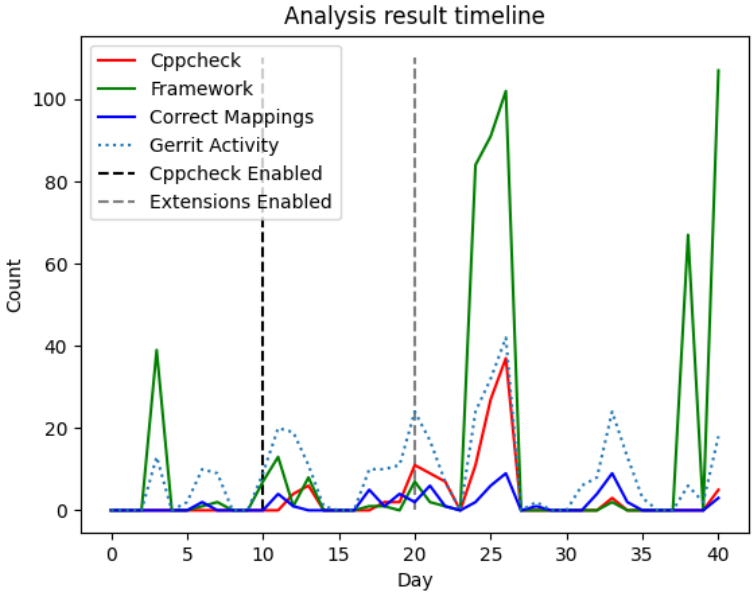

Figure 8.4: The timeline of analysis results while the MEAN system was running on the production tool stack.

Figure 8.4 shows the number of published robot comments per day for every analyzer. It also shows an approximation of the total activity per day in Gerrit for the team where MEAN was running. The activity is measured in number of times MEAN started an analysis, so if MEAN started both Cppcheck and Framework Analyzer for a specific patch set, it is counted twice. This means that the activity will generally be higher after we integrated Cppcheck, around day 10. Also, due to limitations in the data, the figure also includes the times an analysis was started by us when we tested the system by pushing code to the test repository in Gerrit. It can be seen that the spike of activity around day 25 also led to a higher amount of analysis results. On the other hand, the spike of results from the Framework Analyzer just before day 40 was due to one large code change, explaining the low activity measured.

Chapter 9
Discussion

In this chapter, we discuss the results related to the different research questions. In addition to this, we go through some possible future work for MEAN in the last section of the chapter.

9.1 RQ_1: Noise Reduction

We enabled the two modules meant to reduce robot comment noise after MEAN had been running for about 20 days on the production tool stack. The `Visual Feedback Feature`, which grays out 'NOT USEFUL' comments so that developers could 'mentally filter out' those comments. And the `Filter Module`, which filters out comments unrelated to the most recent changed lines. Unfortunately, due to the limited amount of time MEAN was running, together with the fact that the Gerrit activity was relatively low, it is hard to say if these additions are really improvements. As can be seen in Figure 8.3, the 'NOT USEFUL' button was only clicked on 16 comments. This means that it is unlikely that most developers have seen the grayed out comments. As for the `Filter Module`, it is hard to see any effect from it in Figure 8.4. In fact, the largest spikes in analysis results were after the `Filter Module` was deployed. Although it is possible that the spikes would have been even larger without the filtering. On the bright side, the interviews revealed that although the changes had gone unnoticed, the general attitude towards them were positive when it was explained what their purpose was. They thought that it could help reduce the risk of becoming overwhelmed with potentially hundreds of robot comments in one file.

9.2 RQ_2: MEAN Integration

Regarding **RQ_2**, about the integration of MEAN into the Bosch tool stack, we were fairly lucky. Both Axis, where MEAN was original integrated, and Bosch had fairly similar tool stacks, at least from the view of MEAN integration. Both companies uses Gerrit as review

tool, and Jenkins as CI system. This meant to us that we could reuse all original modules from MEAN, with only a small amount of tweaking, to adapt the system to Bosch. Unfortunately, Bosch did not use RabbitMQ as message broker. Even so, we concluded that it would probably be easier to set up a small RabbitMQ server than it would be to rewrite a large amount of the MEAN modules to use another type of message system.

However, the integration of MEAN into the Bosch tool stack was not completely without its problems. As detailed in earlier chapters, Bosch is very strict regarding which Gerrit plugins are allowed. This resulted in a problem, since the MEAN Gerrit Plugin is an important part of MEAN. It would of course be possible to run MEAN without the plugin, but not without rewriting some of the modules. Especially so, since a large part of the MEAN configuration system is received from the Gerrit plugin. The 'NOT USEFUL' button would also be non-present, which would mean that the feedback loop would be broken. Fortunately, we did not need to take these compromises, since another Bosch team with more control over their systems (notably Gerrit) was found, where we could integrate MEAN.

When we asked about the robot comments in Gerrit that MEAN posted, developers found the 'NOT USEFUL' button to be clear and immediately obvious what it meant. Instead, confusion arose as to what the purpose of the "PLEASE FIX" button was for, and when to use it. They believed that it had been added by the MEAN system together with the 'NOT USEFUL' button. This caused users to question which button to press due to the ambiguity brought on by the "PLEASE FIX" button.

9.3 RQ_3: Perceived Value of Analysis

We gathered that the consensus regarding the value of program analysis results was useful. Another value they found in the results was that they assisted the developers by taking away some of the burden with having to detect issues in the code. It was apparent that the Austrian team valued static analyzers by their diligent use of such tools. They also had a dedicated group that fixed the issues with the tools within days if possible. Before starting a new project, they tested out their tools, thus finding and fixing issues with the tools before the developers started working on the project. If they were unable to fix them, they downgraded them to older version or disabled some tools entirely. This was due to the high cost that would be induced if the project in later development stages needed to be paused to resolve tooling issues.

Then, while working on the code, the developers made constant use of their tools before pushing for review with a requirement that no warnings or errors were present. Still, mistakes could happen, unclear issues such as false positives could be postponed to the review phase. Faults not caught by their local tools could be found during the Jenkins verification builds, thus passing through the MEAN system and posted to Gerrit. We thus find it clear as to why the amount of analysis results presented in the graphs in Chapter 8 were so low. Namely, due to tools being fixed before the project starts and their heavy usage of tools before sending for review.

The maintainers of the tools often got requests to disable a certain tool or category of it to avoid having to fix certain problems. In other cases, developers tried to re-trigger the Jenkins builds in hopes of it passing the second time. Sometimes simply contacting the tools department, asking why their builds did not pass. Clearly showing the unwillingness to jump

to another platform to go through large Jenkins logs to find what the issue was. Thus, we found that developers and maintainers alike seemed to appreciate the benefits that the MEAN system brought, thus saving time for several parties. This shows that the value in static analysis results can depend on how easy it is to come by the results, and how clear the message is. This is in line with what Johnson et al. [2] found when conducting interviews with 20 developers, integration problems and result understandability were barriers to utilizing analysis results.

This is where the usefulness of the 'NOT USEFUL' button comes in, developers and maintainers of the tools used, found the purpose of the button easy to understand. The maintainers found that it would save significant time if, by clicking the button, a notification to a configurable list of maintainers could be issued. Thus, pinpointing the issue of what a developer does not like, removing the need to spend time investigating build failures.

9.4 Future Work

In this section we discuss a few possible points for future work in this area.

MEAN Integration As discussed in Section 9.2, both Axis and Bosch uses Gerrit and Jenkins, which made the integration of MEAN fairly smooth. However, there are still several departments which do not use Gerrit in Bosch. As such, it would be interesting for future work to integrate MEAN into a more different tool stack, using another review tool than Gerrit for instance. This would mean that a few MEAN modules would need to be rewritten. It would also be interesting to see how much the modular nature of MEAN actually helps to reduce the work needed to fit a more different tool stack.

MEAN Improvements As detailed in the method, our two MEAN improvements were only running in production for two weeks, due to time constraints. It would be interesting for future work to run such improvements for a longer time. As such, it would be possible to gather more data about how these additions affect the developers and evaluate if they are indeed improvements. Other additions would also be interesting to investigate, such as other kinds of feedback buttons on the robot comments, or automatic enabling/disabling of analyzers based on 'NOT USEFUL' feedback etc.

As mentioned in Section 9.2, several developers found Gerrits "PLEASE FIX" button in the robot comments to be confusing. Thus, a future improvement to MEAN could be to remove this "PLEASE FIX" button, thus highlighting the more clear 'NOT USEFUL' button in more detail. Another topic that came up during the interviews was that the gray color of the `Visual Feedback Feature` was not very compatible with the dark mode in Gerrit. Therefore, a future improvement would be make the feature compatible with this mode.

MEAN Comparison Two research questions that we were considering for this project in the early stages were 'What is the difference in review time before and after the deployment of MEAN?' and 'How does the integration of MEAN affect usage of analysis results for developers at Bosch?'. Unfortunately, these questions were scrapped when the project changed to focus more on MEAN improvements. In addition to that, we were concerned that there the amount of data gathered would be too low for these questions to draw any conclusions.

However, it would still be interesting for future work to put a larger focus on investigating exactly how the integration of MEAN affects the development process.

Chapter 10
Threats to Validity

In this chapter we go through the different shortcomings and limitations of our work, that might affect the generalization of the results.

10.1 External Validity

Short Deployment Period Our original plan was to deploy MEAN into production early, and keep the system running for several months, to gather as much data as possible. However, due to several obstacles, detailed in the method, Chapter 3, MEAN ended up only running on the production system for a little more than a month. This is also in contrast to the original MEAN developers deployment of MEAN at Axis, which was running for a total of 11 weeks [7]. This means that the results obtained might not reflect what would happen if MEAN was deployed for a longer period of time. For instance, it might take some time for the developers to get used to the robot comments, and start to click the 'NOT USEFUL' button. Also, since MEAN was only deployed for one team of about 30 developers, compared to the over 850 active users during the Axis thesis, the results obtained might differ substantially from what would have been obtained from another team at Bosch or another company altogether.

Analyzers Relatively few analyzers were wrapped and running in MEAN, two in-house analyzers and the 'of-the-shelf' analyzer Cppcheck. The in-house analyses were a natural choice to integrate into MEAN, while we added Cppcheck to prevent the risk of too few results being generated with so few analyzers running. This means that the results we gathered are strictly tied to these analyses, and fairly different results can be expected if other analyzers had been selected.

10.2 Internal Validity

Participant Response Bias Interview participants are more likely to give positive feedback to a system developed by the interviewer [33]. This means that the qualitative data might be biased, since we are the authors of the two MEAN extensions. To counter this, we encouraged the interview participants to also give negative feedback, so that we could use that feedback to potentially improve the extensions.

Interviews At first, we planned to perform interviews with about six different people in three different phases. Unfortunately, due to time constraints and difficulty finding willing participants, we could only interview four developers, in one phase. These developers were selected by asking our contact person in the Bosch team for appropriate participants. With such a relatively low number of participants, we cannot be absolutely certain that they are a good representation of the whole team. Therefore, the results might be biased in certain ways. In addition to this, we cannot know what criteria our contact person used to select the participants. It is possible that this selection has introduced additional bias.

Chapter 11
Conclusions

Program analysis is a valuable tool that can be used to improve code quality and reduce the need for manual code reviews. However, it is not without its downsides, such as false positives and difficult to integrate into the tool stack. In this master's thesis, we integrated the open source data-driven program analysis system MEAN into the tool stack of a team at Bosch. While doing this, we discovered that MEAN can be integrated into a tool stack that is slightly different to where MEAN was originally deployed at Axis. However, it was not without its hardships, which we had to solve on the way, some in the form of bug fixes to the MEAN open source repository. We incorporated a mix of in-house analyzers and 'off-the-shelf' analyzers into MEAN.

Unfortunately, we could not utilize the data-driven part of MEAN to tweak or disable analyzers based on the 'NOT USEFUL' feedback gathered. This was because the activity was fairly low in Gerrit, and the interaction with the robot comments via the 'NOT USEFUL' button even more so, to the extent that almost zero clicks were made. However, when Cppcheck was introduced, more 'NOT USEFUL' feedback was generated, indicating that the two in-house analyzers were garnering few 'NOT USEFUL' clicks due to already being well tuned.

After MEAN had been up and running for a few weeks, we deployed a couple of improvements to MEAN we developed to reduce robot comment noise. However, due to time constraints, it is hard to draw any conclusions regarding the effectiveness of the improvements. Also, the interviewed developer that had interacted with the robot comments did not notice a difference. All in all, the answer to the question *"What is the effect of integrating MEAN in the developer workflow at Bosch?"* is that while MEAN was running at Bosch, it had only minor effects on the workflow of the developers.

References

[1] N. Kikuchi and T. Kikuno, "Improving the testing process by program static analysis," in *Proceedings Eighth Asia-Pacific Software Engineering Conference*, pp. 195–201, 2001.

[2] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 672–681, 2013.

[3] N. Imtiaz, A. Rahman, E. Farhana, and L. Williams, "Challenges with responding to static analysis tool alerts," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 245–249, 2019.

[4] M. Nachtigall, L. Nguyen Quang Do, and E. Bodden, "Explaining static analysis - a perspective," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pp. 29–32, 2019.

[5] C. Sadowski, J. Van Gogh, C. Jaspan, E. Soderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 598–608, 2015.

[6] E. Söderberg, "Tricium - tricorder for chromium.." `https://bit.ly/tricium-early-design`, 2016.

[7] A. Ljungberg and D. Åkerman, "Data-driven program analysis deployment," 2020. Student Paper.

[8] A. Ljungberg and D. Åkerman, "Mean - meta analyzer,." `https://gitlab.com/lund-university/mean`.

[9] Robert Bosch AB. `https://www.bosch.se/om-bosch/bosch-i-sverige/lund/`.

[10] Axis Communications. `https://www.axis.com/sv-se/contact-us/axis-experience-center/lund`.

[11] Gerrit Code Review, "Gerrit code review." `https://www.gerritcodereview.com/`.

[12] A. Ljungberg, D. Åkerman, E. Söderberg, J. Sten, G. Lundh, and L. Church, "Case study on data-driven deployment of program analysis on an open tools stack," in *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*, IEEE - Institute of Electrical and Electronics Engineers Inc., 2021. 43rd International Conference on Software Engineering: Software Engineering in Practice, ICSE 2021 ; Conference date: 23-05-2021 Through 29-05-2021.

[13] Dan Radigan, "Continuous integration." `https://www.atlassian.com/agile/software-development/continuous-integration`.

[14] Sten Pittet, "Continuous integration vs. continuous delivery vs. continuous deployment." `https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment`.

[15] Jenkins, "Jenkins." `https://www.jenkins.io/`.

[16] GITLAB, "What is version control?." `https://about.gitlab.com/topics/version-control/`.

[17] git-scm, "Git –distributed-is-the-new-centralized." `https://git-scm.com/`.

[18] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 712–721, 2013.

[19] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at google," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pp. 181–190, 2018.

[20] Gerrit Code Review, "Working with gerrit: An example." `https://gerrit-review.googlesource.com/Documentation/intro-gerrit-walkthrough.html`.

[21] Google, "Containers at google." `https://cloud.google.com/containers`.

[22] IBM Cloud Education , "Containerization." `https://www.ibm.com/cloud/learn/containerization`.

[23] Docker, "Docker." `https://www.docker.com/`.

[24] RabbitMQ, "Rabbitmq." `https://www.rabbitmq.com`, 2021.

[25] Sourceforge, "Cppcheck - a tool for static c/c++ code analysis." `https://sourceforge.net/p/cppcheck/wiki/Home/`.

[26] W. Adams, *Conducting Semi-Structured Interviews*. 08 2015.

[27] C. Robson, *Real World Research*. John Wiley & Sons Ltd., 2011.

[28] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *(ICSE), 2013 35th International Conference on Software Engineering*, pp. 931–940, 05 2013.

[29] M. G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran, "Making defect-finding tools work for you," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, vol. 2, pp. 99–108, 01 2010.

[30] Shipshape, "Shipshape." `https://github.com/google/shipshape`.

[31] "Gnu diffutils (version 3.6, 6 may 2017)."

[32] E. Myers, "An o (nd) difference algorithm and its variations.," *Algorithmica*, vol. 1, no. 1-4, pp. 251 – 266, 1986.

[33] N. Dell, V. Vaidyanathan, I. Medhi, E. Cutrell, and W. Thies, ""yours is better!": Participant response bias in hci," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, (New York, NY, USA), p. 1321–1330, Association for Computing Machinery, 2012.

Appendicies

Appendix A: Interview Guide

Continuous Interviews

- **Type:** semi-structured which means you can clarify things online, within subject design - using the same group of developers.
- **Medium:** Microsoft Teams calls.
- **Length:** 10 to 30 minutes.
- **Goals (What information do we want to get out of this interview?):** Find out about "Impact of integration change"

Type of Questions

- Questions about their role
- Examples of when they found/noticed the difference?
- Value of analysis results in Gerrit.

Interview Guide

Introduction

"Thank you for agreeing to participate in this interview. My name is Michael and this is my colleague Mattias. This interview is to gain insight about the effects the MEAN System has had for you, so that we can better understand how the MEAN System performs at Bosch. Therefore we are interested in your experience with this system and program analysis.

This interview is voluntary and anything said will remain anonymous. What you say during this interview will only be used for our thesis and eventually help improve Software

development at Bosch. This interview will take about 30 minutes. We would like to record the audio of this interview with your permission since we don't want to miss any important details you share with us.

Nothing recorded will be quoted with your name. Is it okay [name of developer], if we start recording the audio now?

Do you have any questions about anything that we have just explained?"

Ice-breakers

- *Do you still visit the office?*
- *How is working from home?*

Background: Developer

What is your role at the company?

Prompts

- *How long have you been working with software development?*
- *How many years?*
- *Which parts of the organization?*
- *Do you have any other background that is relevant in the context of program analysis?*
- *How long have you been working at Bosch?*

Background: Analyzers

Tell us about where you have come across any program analysis results at Bosch?

Alt: Please show us a small example of how you work with the tool stack and how you use analysis results?

Prompts

- *How have you used any of these results?*
- *(If not) Why is that so?*
- *How have analyzers been triggered?*
- *Where have results been published?*
- *What has been the user response?*
- *Where do you find these results?*

Which analyzers have been used at Bosch? (if many, some important ones)

Prompts

- *How has this changed over time?*
- *Has any analyzer been mandatory?*
- *In what scale(local dev/team dev) are they used?*
- *How are in-house analyzers compared to commercial ones?*

Robot Comments from MEAN

What difference have you noticed in Gerrit in reviews, specifically comments from [“Non-Interactive User”]?

- **Prompts when respondent answers "yes"**
- *How do you perceive the comments from [“Non-Interactive User”]?*
- *How do you perceive the comments placed in the diff?*
- *Examples of when you found/noticed the difference?*
- *How often do you read/view them?*
- *How has the new addition changed your workflow when inspecting a review?*
- **Prompts when respondent answers "no"**
- *"Begin by sharing the screen and presenting a scenario with robot comments in Gerrit."*
- *Take a look at this scenario where analysis results in the form of robot comments have been published in Gerrit. What do you think about having the analysis results presented like this?*
- *How do you perceive the comments from [“Non-Interactive User”]?*
- *How would this change your workflow?*

’NOT USEFUL’

What value do you find in having analysis results in Gerrit?

Prompts

- *If you would click ’NOT USEFUL’ on any comment, why would you do so?*
- *(If pressed ’NOT USEFUL’): Which were the main reasons for comments being ’NOT USEFUL’?*
- *How would you like ’NOT USEFUL’ comments to be handled?*
- *(“Replies with not valuable”): Could you elaborate on why you find it not valuable?, What changes would make it valuable?*

Noise Reduction

Have you noticed any difference in the latest two weeks?

Prompts

- *(Answers "no") We added a color change if 'NOT USEFUL' has been pressed, how valuable do you think this addition is?*
- *(Answers "no") We added so that comments are filtered away if they are related to unchanged lines, how valuable do you think this addition is?*
- *(Answers "yes") What changed did you notice? (Continue with above questions)*

Conclusion

What future do you see for the MEAN System at Bosch?

Prompts

- *Why do you think it's (not) possible?*
- *What would be necessary to include MEAN into the workflow at Bosch?*

Finishing with a show of appreciation.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2021-06-10

EXAMENSARBETE The Costs and Benefits of Acting on Program Analysis Results
STUDENTER Mattias Leifsson, Michael Pater
HANDLEDARE Emma Söderberg (LTH), Ali Houmani & Robert Lagerstedt (Robert Bosch AB)
EXAMINATOR Görel Hedin (LTH)

Värdet av programanalys

POPULÄRVETENSKAPLIG SAMMANFATTNING **Mattias Leifsson, Michael Pater**

Automatiska analysverktyg är användbara för att försäkra sig om att programkod håller hög kvalitet. Men problem som falsklarm och låg användarvänlighet gör verktygen svåra att använda. MEAN (MEta ANalyzer) är ett system som försöker åtgärda några av dessa problem.

MEAN är ett system med öppen källkod som är byggt för att underlätta införandet av automatiska analysverktyg i utvecklingsprocessen. Bilden nedan till höger ger en överblick av hur MEAN fungerar. När en mjukvaruutvecklare gör en kodförändring, så skickas den automatiskt till MEAN, som startar de automatiska verktygen som analyserar koden. Resultaten av analysen skickas sedan tillbaka till utvecklaren så att denne lätt kan ta del av resultaten och justera sin kod baserat på resultaten. Om utvecklaren anser att ett resultat är fel, till exempel om det är ett falsklarm, så kan denne lätt ge återkoppling till MEAN genom en 'ej användbart' knapp. Denna återkopplingen sparas i en databas, och kan sedan användas av MEAN underhållare för att ändra konfigurationen av systemet utifrån återkopplingen.

I vårt examensarbete tog vi MEAN och integrerade det till ett team med ungefär 30 utvecklare på Bosch. Det innebar att vi behövde göra en del ändringar och tillägg till MEAN för att anpassa det till utvecklingsprocessen på Bosch. Dessutom bidrog vi med några buggfixar till MEANs öppna källkod som vi stötte på under integreringsprocessen. Efter att vi låtit systemet köra på Bosch i några veckor, så rullade vi ut två förbättringar till systemet, för att se hur stor skillnad det blev i beteende för och efter förbättringarna.

Utvecklarna var positivt inställda till att resultaten från verktygen var mer åtkomliga medan de som underhöll verktygen uppskattade 'ej användbart' knappen. De ansåg att det kunde spara tid både genom att resultaten blev lättare för utvecklarna att se, men även effektivisera processen av identifiering och lagning av fel i verktygen. Resultatet indikerar att MEAN är ett system som många olika företag kan tjäna på att använda.