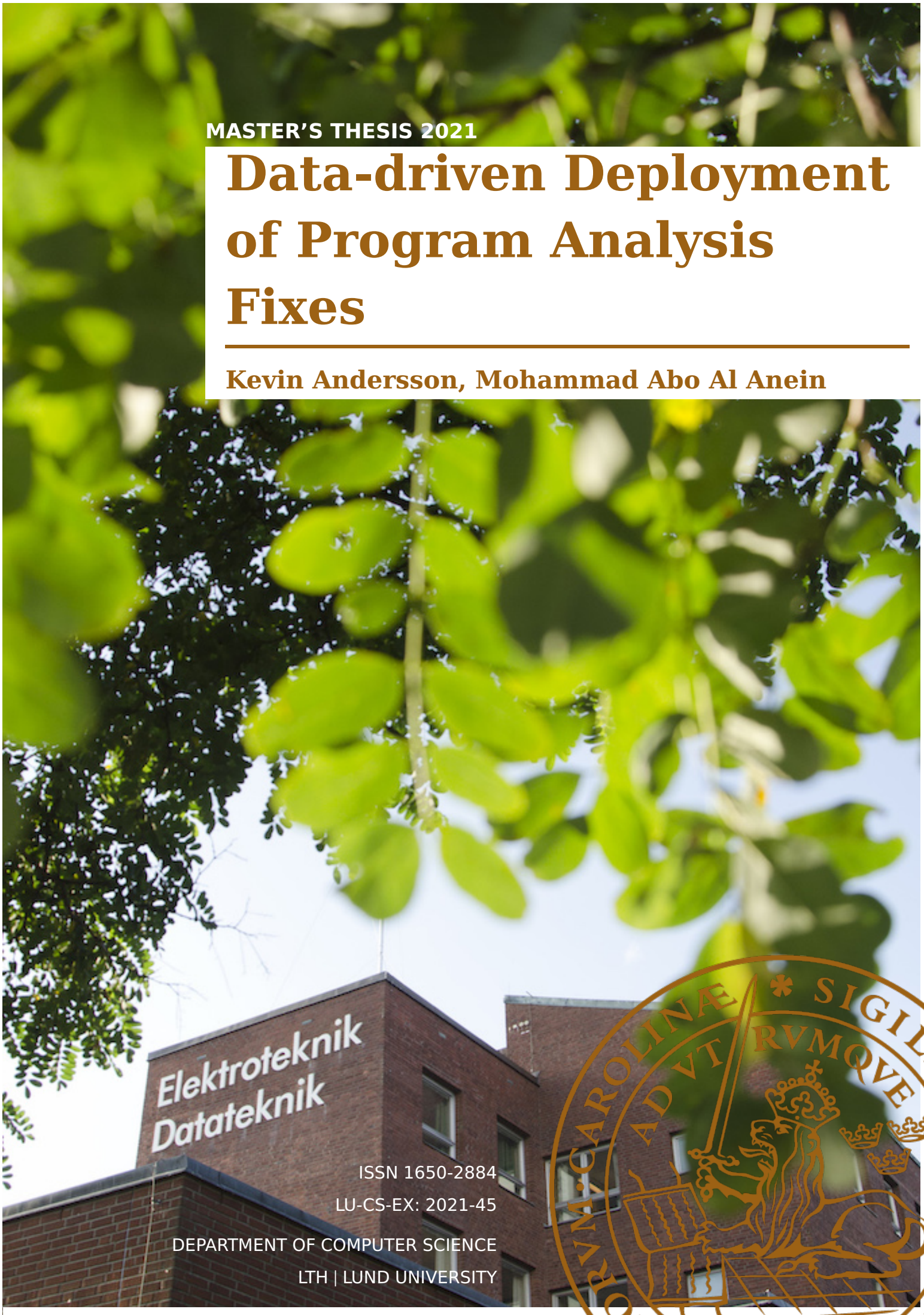


MASTER'S THESIS 2021

Data-driven Deployment of Program Analysis Fixes

Kevin Andersson, Mohammad Abo Al Anein

ISSN 1650-2884

LU-CS-EX: 2021-45

DEPARTMENT OF COMPUTER SCIENCE

LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2021-45

Data-driven Deployment of Program Analysis Fixes

Kevin Andersson, Mohammad Abo Al Anein

Data-driven Deployment of Program Analysis Fixes

Kevin Andersson
dat13kan@student.lu.se

Mohammad Abo Al Anein
mo3541ab-s@student.lu.se

October 19, 2021

Master's thesis work carried out at Axis Communications AB.

Supervisors: Emma Söderberg, emma.soderberg@cs.lth.se
Jon Sten, jon.sten@axis.com
David Åkerman, david.akerman@axis.com

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

Abstract

Program analyzers can be really helpful in developing code, helping to find otherwise unnoticed issues and helping developers avoid common mistakes. However, using analyzers often come with needing to handle false positives, confusing warning messages and configurations that are specific to each analyzer.

To counteract these issues, a data-driven program analysis system can be used to enable continuous tuning during run-time. One such recently developed system is called MEAN, short for MEta ANalyzer, a relatively new system that is actively in development. This means that the program might be missing features in comparison to other data-driven program analyzers, one of these being the ability to provide fix suggestions.

In this thesis, the goal is to investigate how a fix suggestion feature might best be implemented and tested in MEAN. This fix suggestion feature will provide MEAN with the capability to not only send messages, but also to send specified suggestions that can automatically be integrated into the code. Furthermore, this thesis will require some deployment of MEAN with these new features in order to evaluate how well the new changes are received.

The conclusions reached in this thesis is that the new feature is well received, but could be implemented better, that is, while the use of the feature is easy to understand, the actual process of using the new feature might be unnecessarily clunky. Overall, the addition of fix suggestions appear to have improved the interaction, and provides developers with a quick and easy way of integrating suggestions into their code.

Keywords: Static Analysis, Data-driven, Program Analysis, Fix Suggestions

Acknowledgements

We would like to thank Emma Söderberg, Jon Sten and David Åkerman for their active supervision of this thesis.
We would also like to thank interviewees, participants of the survey and users of MEAN for their contributions to this thesis.

Contents

Chapter 1

Introduction

Program analyzers, a kind of software which is used to find issues and potentially also provide fixes for issues in code, have been available for a long time. Lint is one of the earliest program analyzers, which was initially released as a Unix utility in 1978, and was so influential that the word "linter" has come to be used for static code analysis tools. They are more lexical or syntactical in nature [1]. Over time, analyzers have received advanced features in order to provide more assistance to developers. One such feature is the inclusion of fix suggestions in analyzer messages, extending the basic functionality of analyzers to find and warn of issues to also include code that might solve these issues.

A number of studies over the last years have shown that using program analyzers can lead to a lot of benefits, but also inhibit these benefits with a lot of usability issues, limiting the extent to which these tools are actually used. Examples of these issues include delivering an exceedingly large amount of issue reports, reports that are hard to interpret, a lot of false-positives, as well as often being poorly integrated in the overall workflow [2][3]. If these drawbacks could in some way be mitigated, it could potentially be of interest to a lot of developers. The analyzers could with these changes enable more secure and correct code by finding issues that might otherwise be missed, minimizing required maintenance in later stages of development and thereby potentially saving a lot of time.

One of the ways these issues can be mitigated, is through the use of data-driven tuning powered by developer feedback. By collecting feedback and letting this data shape the behavior of the analyzer, analyzer maintainers can

become aware of and counteract aspects they deem problematic. One of the ways to design such a system is through the meta-analyzer system approach. A meta-analyzer system is a type of software/framework, which given a segment of code, collects issue reports from a collection of program analyzers (see Figure 1.1). This output is then sent to the developers through a common interface, and finally the system receives feedback from the developers, which is used to configure the output sent for later code segments. This allows the developers to have a direct input on the output generated, which in turn leads to the meta-analyzer generating more relevant output. Several meta-analyzers have been developed over the years, including Tricorder [4], Shipshape [5] and Tricium [6].

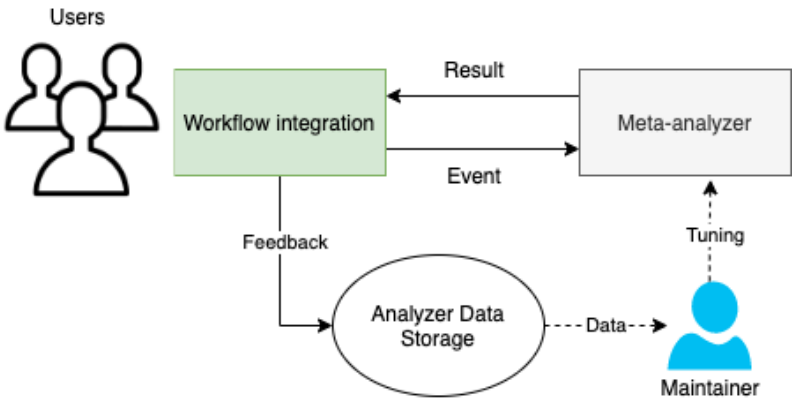


Figure 1.1: An overview of a meta-analyzer system.

MEAN is another of these meta-analyzers, and is the meta-analyzer which will be the core of this thesis. The MEAN meta-analyzer was built by Åkerman and Ljungberg as part of their masters thesis [7], and this report was later extended in Ljungberg et al. [8].MEAN was first deployed at Axis Communication AB during 2020, for a period of 8 weeks, and led to several categories of errors being deemed superfluous [8]. Axis Communication AB wishes to continue the development at MEAN in order to improve the system's usability and precision. Furthermore, work has also been done at Bosch to integrate the MEAN system [9], which means that potential development on the system will be all the more relevant to the field of meta-analyzers as a whole (more description about MEAN in Chapter 2.6).

1.1 Objectives

The objective of this thesis is to implement fix suggestions in the MEAN system, allowing the system to forward suggestions from analyzers that would otherwise be ignored. These implementations also need to be evaluated, leading to the creation of three research questions:

RQ1 How do developers act on fixes suggested by robot comments?

RQ2 How can the process of acting on fixes suggested by robot comments be improved?

RQ3 How well can integrated fix assistance improve the process of acting on suggested fixes?

The research questions (RQ:s) are set up in such a manner that they can be tackled in order, where RQ1 corresponds to a first implementation of fixes. RQ2 and RQ3 follow from the improvement of this first version, where RQ2 corresponds to the actual changes made while RQ3 regards the analysis of how well these improvements performed.

1.2 Approach

Structuring the work around the previously mentioned research questions, the work presented in this thesis started with a preparatory phase in order for the authors of the thesis to gain an understanding of MEAN. This was followed by two separate deployments, with separate preparations for each.

For the First Deploy, the goal was to answer RQ1. As such a basic fix suggestion implementation needed to be introduced, and a program analyzer supplying fix suggestions needed to be integrated. The analyzer chosen for the First Deploy was Black. An evaluation was then performed to investigate possible improvements, answering RQ2.

For the Second Deploy, the goal was to answer RQ3, and as such the improvements discovered after the First Deploy needed to be implemented. In order to test the possibilities of incorporating fix suggestions from other analyzers, a new analyzer also had to be integrated. The analyzer chosen was Clang-tidy. This meant that during the Second Deploy two analyzers issuing fix suggestions, Black and Clang-tidy, were used.

After the Second Deploy data once again needed to be evaluated to discuss the findings for RQ3, whereafter the thesis was concluded.

1.3 Delimitations

The subject of this thesis is the introduction of fix suggestions, and as such we will make a concerted effort to work with changes which directly concern fix suggestions or which indirectly helps us implement fix suggestions. Improvements to MEAN outside of this category are considered out of scope, and as such are not implemented.

Furthermore, when possible, the changes made should also be based on the needs of Axis in order to make the project more relevant for the company, as well as making deployment and integration easier for testing.

1.4 Risk Analysis

Prior to the start of the project, we performed

1.4.1 Risks

1. Scope is made too big for the time available
2. Small amount of data for evaluation
   (a) System used for only a small amount of time
   (b) System deployed late
   (c) System adopted late because of improper introduction
   (d) Unfamiliarity with the original system might lead to irrelevant data, as they might have comments regarding features not implemented during this thesis.
3. Technical issues (overreliance on technology for communication as an effect of corona)
4. Too long lead times for creating a good implementation of fix suggestions

1.4.2 Risk Mitigation

1. The focus is first and foremost to create a baseline for testing. More features/changes will be added after the first test phase based on feedback received. Meetings will be had regularly (once a week) with supervisors to discuss what might be good additions to the system.
2. (a) Find a team early to ensure availability, this ensures a larger sample size. Also try to get a team which is interested in the chosen analyzer, to ensure that the tool is used as part of the workflow. Using the team/teams involved in the previous thesis will probably speed this process up as well, as they have an understanding of what MEAN is and as such will be more proficient with the system.
   (b) Hopefully avoided by not going beyond a minimum viable product.
   (c) Request access to needed infrastructure early in order to not have waiting delays

(d) Find a team early, as described in (a). Also hold a presentation of the system before deployment to increase familiarity

(e) As mentioned in c., increase familiarity to mitigate irrelevant data. Also, tailor scenarios to coincide with the features introduced in this project.

3. Test communications early, and consider alternative communication channels if the existing channels are not fit to task.

4. Create runnable versions for each milestone in the timeplan dedicated to the creation of new features, in order to ensure that progress is made.

5. Make concrete suggestions for changes before working on changes in order to avoid starting unnecessary work on irrelevant features.

6. Pinpoint the points of criticism in feedback in order to narrow down changes which are of interest.

1.5 Glossary

Alert Notification of a potential error in a program

API Application Program Interface

AST Abstract Syntax Tree

Change A submission of modified code uploaded on Gerrit

CI Continuous Integration

Config Configuration file

Finding An issue found in the code

I/O Input/Output

IDE Integrated Development Environment

OSS Open Source Software

Tool alert Message sent from analysis tool, usually referred to as a warning (does not need fixing) or error(needs fixing)

Fix Solution to erroneous code

Fix suggestion A suggestion for a fix

RabbitMQ An open-source message-broker software

Chapter 2

Background

In software development there exists a wide variety of tools to make work easier and more efficient, where MEAN is one of these tools. As this thesis will aim to expand on MEAN, it is important to talk about general concepts that MEAN touches on. This section will therefore briefly cover the following concepts:

- Code Review System
- Continuous Integration
- Virtualization
- Program Analysis

For each of these concepts, further detailing of the concepts will be provided when necessary.

2.1 Code Review System

Code review is an activity done to among other things find bugs in code, improve quality of the code and to allow knowledge to be transferred between developers through the review of each others code [10]. There are two different approaches to code review, traditional and modern code review. The difference between them is that in traditional code review developers schedule a meeting in order to review a set of code synchronously. In modern code review this process is instead performed asynchronously through the use of code review systems. Code review systems may use different approaches to enable code review, which may include being built on top of a version control

system. A version control system (VCS) is a system that keeps track of modifications of code, which makes it possible to go back to a previous version of the code. A revision is a term used in VCS that describes a set of changes made to a list of files [11].

The process of code review is that the author makes a code-change and the reviewer gets added and notified and adds his/her comments. Thereafter the author responds and updates the code, the reviewers look at the code again and then the author respond again and so on until the code is approved and submitted.

2.1.1 Gerrit Code Review

Gerrit is a web-based code review system currently in use at Axis, and is built around a VCS called Git. Developers can review each other's code using Gerrit, using a wide variety of features to facilitate productive discussion of code. This includes the ability to score uploaded changes.

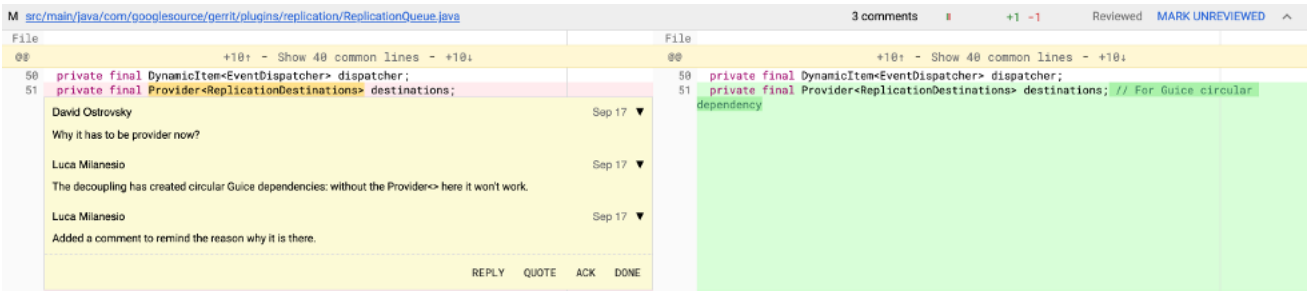


Figure 2.1: Code commenting in Gerrit

There are five labels which span the score scope that are given by reviewers.
Review labels are:

2 "Looks good to me, approved."

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

0 "No score"

-1 "I would prefer this is not merged as is"

-2 "This shall not be merged."

Important to note is that this is just how Gerrit suggest the labels be used, and they won't necessarily have the same meaning in every project.

In order to enable reviewers to use these labels and comment on code however, Gerrit provides a framework for developers to upload their changes to

a server (see Figure 2.2). The workflow used to interact with this framework goes as follows:

1. A developer creates a change, making him an owner.

2. Developers can then start working on this change, writing and uploading code in the form of a patchsets (patchsets represents individual revisions of a change) to the change on Gerrit, as well as adding reviewers/CC:s to the change.

3. Reviewers and CC:s add comments to the code if there are some things that might be improved.

4. These comments are sent along with a review label indicating the reviewer's opinion on the change in general

5. The developers read the submitted comments and either change the code depending on these comments or disagrees and gives a reason by responding to these comments

6. The code is updated by the developer and submitted as a new patchset

7. Steps 3 through 6 will be reiterated until the change is approved and submitted, or the developers decide to abandon the change.

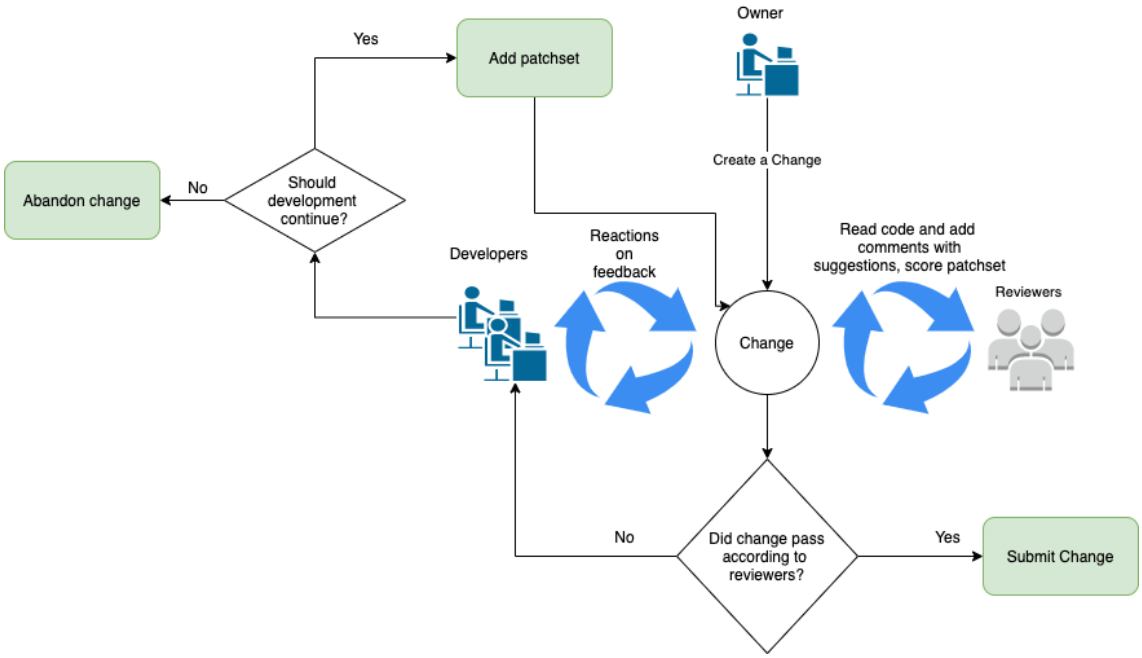


Figure 2.2: An overview of Gerrit process.

2.2 Continuous Integration

With continuous integration, code is integrated continuously in regular intervals or when the code is changed. This integration can be done in wide variety of forms, such as testing the latest version of the code in an isolated environment, code analysis and/or building and delivering the program to a repository. In this way, we verify that the code still works and that the new code or change made by others developers does not break the system.

2.2.1 Jenkins

Jenkins is a open source service where can build, test and check code automatically. Jenkins jobs can be configured to trigger in reaction to a variety of actions, such as when a new code has been pushed to or reviewed on the repository. Jenkins has access to many community-made plugins, providing developers with a lot of flexibility when designing builds within Gerrit [12].

2.3 Virtualization

Virtualization is the idea that a piece of hardware (i.e. a computer) can be made more flexible by adding an isolation layer between application and its environment. There are several methods to do so, we will be mentioning two: Virtual machine managers and containerization [13].

2.3.1 Virtual Machine Manager

A virtual machine manager, also called a hypervisor or VMM, is a virtualization implementation which allows for the creation and management of virtual machines. Virtual machines in turn are execution environments which are either running directly on hardware, or within the users operating systems. These virtual machines provide a separate execution environment by acting as a separate machine with its own OS, libraries, applications and files. This allows the system to, while providing a separate work environment, to utilize a wider array of applications and features. Since a lot of these are dependent on the OS used [13].

2.3.2 Containerization

Containerization is an OS-level virtualization technique that allows the user to isolate applications within a running process on the computer, which consists of an environment containing the application to be used, as well as the dependencies necessary for the application to run. This environment is commonly referred to as a "container". What this means is that in comparison to the

VMM, the container is not reliant on the use of a separate operating system, leading to lower storage and performance requirements [13][14].

Docker

Docker is a set of products bundled in the form of platform as a service, which uses containerization as a way to isolate processes. Docker calls these containers "images". These images can be set up using "Docker files", which are executable files which can be used to specify what these images should contain, and how the image should run by default. As such these images can contain ready-made software, as well as prepared workspaces.

2.4 Publish/Subscribe Message Handling

Publish/Subscribe message handling (or the publish-subscribe pattern), is a method of communication between processes, applications, etc. This method of communication uses a common event service (also called topic and event handler) to enable communication between publishers, processes which submit messages to a communication channel, and subscribers, processes which consume these messages, reading and removing them from the communication channel. This enables decoupling of publishers and subscribers, allowing for processes to in large part be independent from one another [15].

2.5 Program Analysis

Program analysis can be explained as the process of automatically diagnosing and finding potential issues in the code of a program. Program analyzers can be divided into two categories: static program analyzers and dynamic program analyzers. Static program analyzers parse through the files to find errors, as in they don’t actually run the program.

Dynamic program analyzers are analyzers that analyze the program while it is running. Actually running the code might lead to catching run-time errors that are impossible to find in the source code alone.

2.5.1 Black

Black is an open-source Python code formatter first released in 2018 under the MIT license [16] created by Lukasz Langa [17]. When operating without additional flags or options, this python formatter will take a python file as input and reformat the code in the file, directly applying these changes in accordance to a standard specified by the developers of Black.

Black, while still in its Beta phase, is used in several open-source projects such as SQLAlchemy and pytest, and is used within several organizations including Facebook, Mozilla [18] and Axis.

2.5.2 Clang-tidy

Clang-tidy is an open-source static code analyzer for C++ based on the Clang project, a language front-end and tooling-infrastructure for languages such as C and C++ [19]. Clang-tidy checks for multiple types of errors, including style-violations and bugs, based on what checks have been enabled/disabled by the developers, and also allows these checks to refactor code at the users digression. Clang-tidy is developed to be highly modular, and as such enables the integration of developer-made checks into Clang-tidy [20].

2.6 The MEAN System

MEAN is a meta-analysis-system that runs program analyses automatically and makes it easier for developers to get the comments and suggestions for their code from the analysers. Results from analyzers are presented in the form of robot comments.

A brief summary of how MEAN works (see Figure 2.3):

1. The author pushes their code to Gerrit as "review".
2. A change event is sent to MEAN publisher.
3. MEAN publisher sends a *mean request* to main system.
4. An *analyze request* is sent by main system to the analyzer executor.
5. The analyzer executor runs the code through one or several corresponding analyzers depending on configuration.
6. The analyzer executor then sends the analyzer results to the main-system in the form of an *analyzer event*.
7. The main system sends a *MEAN event* to the Robot publisher component. The Robot publisher then sends robot comments to Gerrit by using the Gerrit REST-API.
8. The reviewers and author get the results of each analyzer as robot comments where they can report specific comments as not useful or fix the code according to the comments.

All of the messages sent use the same external messaging client RabbitMQ to transfer messages between different MEAN components.

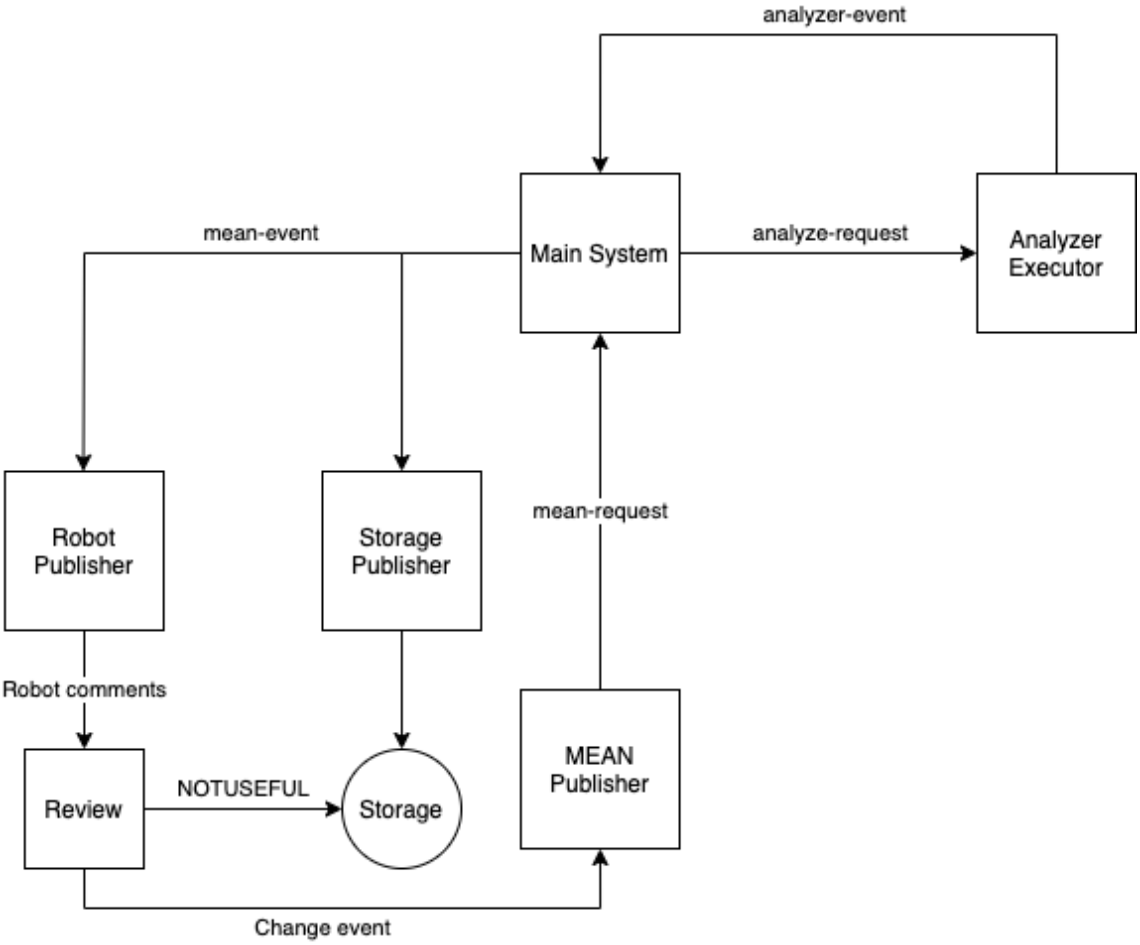


Figure 2.3: Overview of the MEAN system.

2.6.1 Robot comment

Robot comments in MEAN consist of three part description, category field and two buttons PLEASE-FIX and NOT-USEFUL (see Figure 2.4). The category is the check used by the operating analyzer. When sending robot comments by using Gerrit REST-API, PLEASE-FIX button appears automatically while NOT-USEFUL button has been added as plugin for robot comments in Gerrit. If a reviewer clicks PLEASE-FIX button then he/she indicates to the developer/s that this comment should be fixed while by clicking NOT-USEFUL button it means that the developer (reviewer or owner) sees that this check or comment is not useful, for example this check has false positive.

Figure 2.4: An example of a robot comment from MEAN.

2.6.2 The MEAN Publisher

The MEAN publisher is the first micro-service triggered when a new patchset is published, whose purpose it is to forward messages from Gerrit to the MEAN main system. This micro-service consists of two parts; a Jenkins job and a python script. The Jenkins job is run whenever a patchset is created in a repository monitored by the job, which in turn runs the python script to gather additional information required to reply to the patchset. This information is then forwarded in the form of a MEAN request to the MEAN main system.

2.6.3 The MEAN Main Component

The MEAN main service can be thought of as a hub for all the micro-services in use, which is to say that the main responsibility of the service is to receive messages from and deliver messages to other micro-services. Simply put, the main component can be thought of as a hub for the other components. However the MEAN main component also has the responsibility of tracking the current state of MEAN, as well as deciding which analyzers to run based on the configuration settings. The main component sends two types of messages:

- Analyzer-request: Sent to the analyzer-executor job to start the analysis of submitted code. Includes information such as what files should be analyzed, enabled analyzers, and more.

- MEAN event: Can be thought of as status update messages, sent every time the main component receives a message or sends an analyzer request. Based on the message the component reacted to, different information might be sent in these messages. One use of this feature, aside from being able to monitor MEAN, is that the info sent when reacting to analyzer-events can be forwarded to the robot publisher, providing the necessary info to create comments in Gerrit.

2.6.4 The MEAN Analyzer Executor

MEAN analyzer executor is written in a Jenkins pipeline script running on Jenkins. When an *analysis request* is received from the RabbitMQ queue, this request is written to analyze_request.json in the mean/input directory, a message is sent that this analyzer will start analyzing code and the code will be put in mean/code directory. Finally run the analyzer and check that it was successful and the answer (timeout or an error happened) is sent to the main system [7].

From the analyzer executor to main system a message is sent if a state change occurs in the analyzer executor. This message is the analyzer-event which contains information of which MEAN-request this is a response to, as well as information such as which analyzer triggered the message, which file, as well as information of the current status of the analysis. If the message is labeled as a result, it will also contain notes on any errors found through analysis, which will be forwarded through the main component to the robot publisher.

2.6.5 The MEAN Robot Publisher

In order to send the output from MEAN to Gerrit, the robot publisher micro-service is used. This micro-service receives AnalyzerEvents from the MEAN main system, transforms the input and sends the output to the Gerrit plugin through the RabbitMQ server. Going more in-depth, after receiving an AnalyzerEvent, the robot publisher starts with checking whether the repository specified in the AnalyzerEvent has requested comments to be published. If enabled is set to false, the event is sent to a "not-published" queue in order to allow for later access of the information, for example for logging purposes, and no further action is necessary.

If it is set to true, then the next step is to extract the information from the AnalyzerEvent and start creating comments, that conform to the Gerrit API. In this process some missing values might also be set or modified to ensure that functional comments are being sent. These created comments are then transformed to fit the format used by Gerrit, which is then sent to the Gerrit-Server. This data is also sent to RabbitMQ queues for the sake of logging.

2.6.6 The MEAN NOT-USEFUL Server

The NOT-USEFUL Server listens for HTTP requests sent from the NOT-USEFUL button. The data is extracted from the request and sent to a RabbitMQ queue.

2.6.7 The MEAN Gerrit Integration

The MEAN system integrates with the code review system Gerrit. This integration is based on the method used by the Tricium plugin [6]. Gerrit plugin contains a NOT-USEFUL button that allows users to give feedback on whether they deem a comment to be unnecessary, incorrect or otherwise not useful.

Also part of the MEAN Gerrit integration are the configuration files. For each project intent on using MEAN, a configuration file is needed. This file can either be located in the projects configuration branch or a configuration branch belonging to the parent of that project. The file is used to specify whether MEAN should run on this project, as well as what analyzers and flags/checks should be used when analyzing code in the specified repository. As such, these files may be used to not only configure MEAN, but also to potentially opt out of using the system.

Chapter 3

Related Work

This chapter will address the work related to this thesis in order to give more of a background to the thesis, as well as detail what work we may have used as inspiration. The studies/reports mentioned here will mainly be sought after in three ways: Searching with Google scholar/LUBSearch, looking through the contents of relevant conference papers, and utilizing the sources used in the thesis work that this thesis is built on.

3.1 Program Analysis

Program analysis is a wide field, with a lot of different tools and applications, but, as explained in the background section, it is a way to automatically analyze program code. In this section three fundamental questions will be posed and answered; How are program analyzers used, what are the benefits and issues of using program analyzers, and what should an analysis tool provide.

3.1.1 Use of Program Analysis

Program analysis can be used in many different contexts within code development. In a report by Vassallo, Carmine and colleagues [21], code development is divided into three defined contexts: Local programming, Code review and continuous integration. Local programming is defined as the environment in which the programmer is writing the code, as in an editor, where tools can give feedback during coding. Code review is the environment in which developers and automated tools can provide insight on the submitted code. Continuous integration is the environment in which code is automatically tested and ana-

lyzed after being sent to a repository but before actually being incorporated. As part of this analysis, different program analysis tools may be used.

The same report [21] also talked about what kind of errors are likely to be prioritized and fixed in general and for the three different contexts. In general, it is concluded that developers prioritize fixing issues described as more severe by the analyzers and following the policies the development team has agreed upon, among other things. Within each context the issues most likely to be fixed are the following: For local programming it is structure and logic issues, for code reviews the issues are style convention and redundancies, and for continuous integration the issues most likely to be fixed are issues concering error-handling and logic. The report also mentions that even if developers prioritize different errors based on context, the analyzers used aren't configured to converge with these prioritizations, and are in actuality mostly configured once during a projects kickoff.

3.1.2 Benefits and Issues in Program Analysis Use

So what are the reasons for using program analysis tools? Johnson et. al., through the use of interviews with 20 participants, found among other benefits: less costly than bug-searching through more manual processes, enabling enforcement of project standards, and by allowing the developers to catch issues early, and a smoother development process [2]. In another report authored by Do, Lisa Nguyen Quang and colleagues, surveying developers on why they use static analysis tools, it was found that some of the reasons why these developers use static analyzers are because it enables them to code faster and better, as well as using them because of team policies [22].

However, there are several reports that mentions issues associated with the use of analyzers. Johnson et. al. found that issues that might be encountered includes the erroneous reporting of false positives, a lot of tools missing necessary customizability, and having no good way to share settings between multiple developers to allow for easy collaboration [2]. In another report, Imtiaz et. al. sort through a previously assembled dataset to find common grievances when working with static analysis tools, where they find the most common issues to be not being able to/not knowing how to properly filter and ignore tool alerts, as well as being unsure whether a certain alert is a false positive [3]. In another report surveying Microsoft developers on their wants and needs from analyzers, the authors Christakis, M and Bird, C find that some of the most regular problems developers note are that the wrong checks are on by

default, as well as the warnings issued being bad [23].

3.1.3 What Should an Analysis Tool Provide

So what are users looking for in present analysis tools? The easy answer would be a tool that offer all the benefits previously mentioned, with none of the issues. However, it isn't that simple. A study by Nguyen Quang Do, Lisa and colleagues come up with 10 recommendations for static analysis tool features: build your tools around time constraint, make sure that the tool is responsive, allow developers to feed data to the tool, improve explainability of warnings, give personalized warning suggestions fit to the developer, implement collaboration features in the analyzer tool, point developers to helpful practices when blocked by warnings, use differing analyzers to catch different types of issues, use a common report interface if multiple analyzers are used, and introduce policies for analyzers to be used in a positive manner [22].

Johnson, B and colleagues wrote a paper [2] investigating why static analysis tools might not be used. In this paper they find among other things that a quick fix functionality may increase usage, especially if the quick fix is interactive. A downside is that it might become too convenient and the developer "quick fixes" incorrect fixes.

They also mention that giving more context for a given error, for example pointing out when an assignment first takes place when issuing a null-pointer warning, might help a developers more easily parse the relevant code. Finally, they find that developers prefer to have their tools be well integrated into their workflow, specifically in a way as to make the process of using and configuring the analyzer more automated [2].

3.2 Data-driven Program Analyzers

Aside from MEAN there exists a plethora of meta-analyzers, some of which directly influenced the design of MEAN. In this section, three data-driven program analyzers, which may be considered as spiritual predecessors, will be presented in order to give a more complete picture of what MEAN is, and what other tools it may be compared to.

Tricorder

The Tricorder system [4], developed at Google and put into production July 2013. It is a closed source system that uses multiple analyzers to give developers feedback during the code review process. This feedback comes in the form of comments and can be interacted with using four buttons:

- NOT USEFUL: Used to report a bug with the comment
- PREVIEW FIX: Provides a window showing the diff between the original code and the code with the comments suggested fix. The button is only available when a fix is suggested.
- APPLY FIX: Apllies the suggested fix. Only available if there is a suggested fix.
- PLEASE FIX: Creates a review comment asking the developer to fix the previously mentioned comment. This button is only available to reviewers.

As for the analyzers, these are run in three different phases called FILES, DEPS and COMPILATION, which provide information in the form of the actual file to be analyzed, build dependencies and an AST for the entire program respectively.

Tricorder was later followed by two systems aiming at providing Tricorders benefits in a new context.

Shipshape

Shipshape [5], developed shortly after the publication of the Tricorder paper, was an attempt to translate Tricorder to an open source setting. Shipshape changed the design to use less analysis phases, PRE-BUILD and POST-BUILD, and bundled analyzers in Docker containers that implenment a remote procedure call API. Shipshape however has no inherent way to gather NOT USEFUL data. Shipshape was last updated August 2016 and has since been archived.

Tricium

Tricium [6] was prototyped in 2016, and can be seen as the Chromium project version of Tricorder, the Chromium project being the code base for the Google open source browser Chromium. Tricium makes several changes to the system, one of the larger differences being that analysis phases are now specified in a configuration file, and can be modified at will. Another very important change is that Triciums design was made to suit the Chromium project infrastructure specifically, making it less useful as an open source alternative.

Chapter 4
Method

For this thesis, the work can be split into three parts: Preparation in the form of reading up on MEAN and reviewing of related work, implementing the MEAN fix extension, and testing/interviews with developers in order to get feedback on how the features should be designed. The last two steps will be iterated twice, the first time to create an initial solution for testing, and the second time in order to perform a design intervention where improvements will be made to the introduced features, for evaluating the final product and for investigating potential future improvements. With this in mind, we decided on the following workflow:

1. Preparation:
    (a) Reading up on MEAN.
    (b) Local Test setup.
2. Introducing new features and analyzers
3. First Deploy, first deployment of MEAN:
    (a) Interviews and observations.
    (b) Data analysis.
4. Design intervention
5. Second Deploy, second deployment of MEAN:
    (a) Interviews and observations.
    (b) Survey.

(c) Data analysis.

All the steps listed here will be given a shorter explanation in order to give an idea of what they mean in the context of this thesis. The literature review will be excluded, as it has already been discussed in Chapter 2. A swim lane diagram was also constructed to show how the work would be structured (see Figure 4.1).

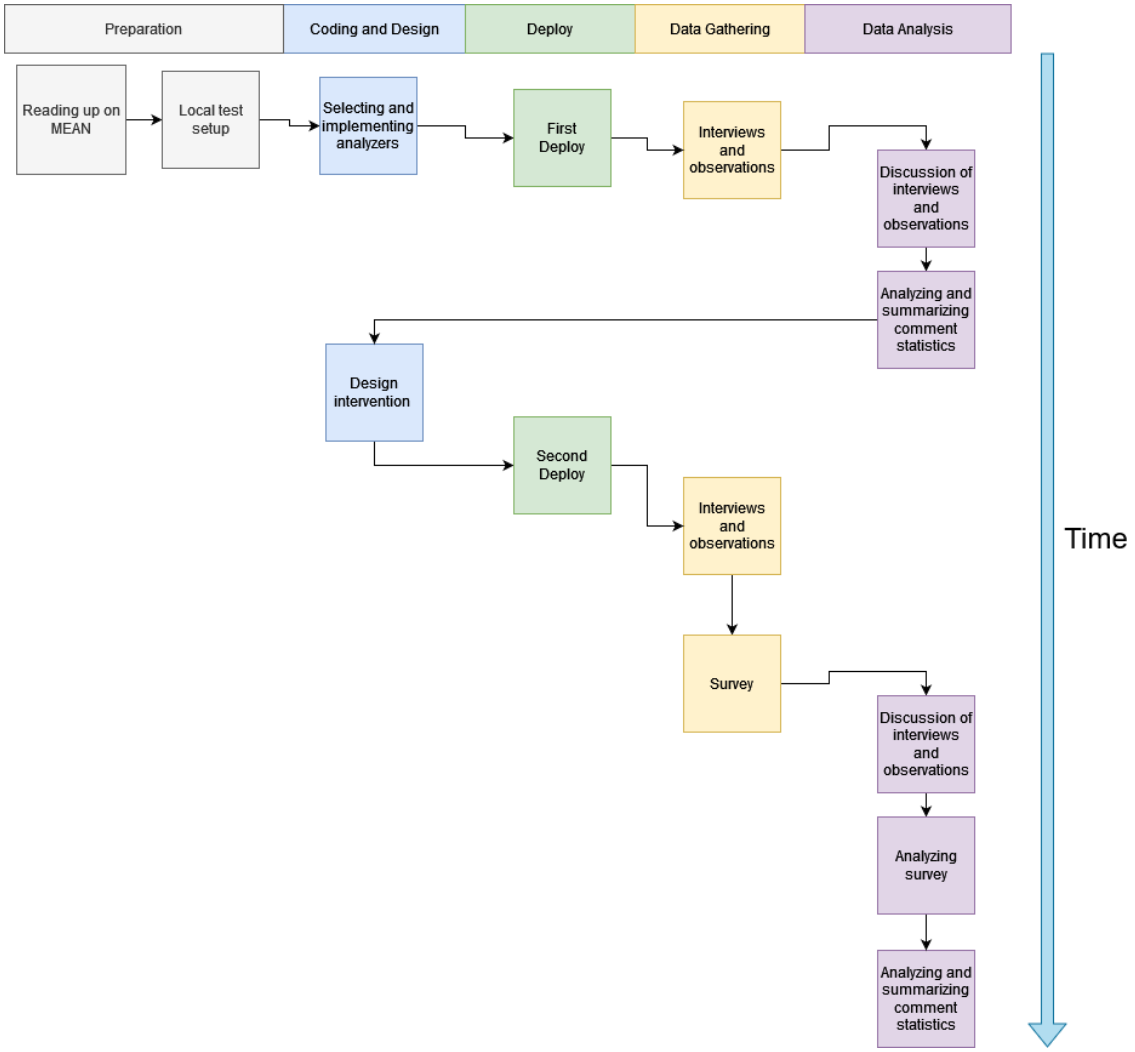


Figure 4.1: An overview of the steps taken in this thesis to answer the proposed research questions. The steps detailed further down in the diagram were performed later in the thesis work.

4.1 Data Gathering

To get an answer to research questions, quantitative and qualitative data has been collected. The quantitative data has been collected using a data visual-

ization tool while the qualitative ones of interviews and surveys. The quantitative and qualitative data is further divided into objective and subjective data. The data is considered as objective if it is dependent on facts or hard data, while subjective is opinion-based data. The objective quantitative data is the collection of e.g., NOT-USEFUL clicks, number of developers who interacted with MEAN and number of comments produced, while the objective qualitative data is analysis of erroneous actions. The subjective quantitative data is the linear scale questions while the subjective qualitative data are interview questions, observations and survey (see Table 4.1).

RQ	Objective/Quantitive	Objective/Qualitative	Subjective/Quantitive	Subjective/Qualitative
RQ1	Collection of NOT-USEFUL click	Analysis of erroneus actions	Linear scale	Interview question Observations
RQ2	None	Analysis of erroneus actions	None	Interview question Observations
RQ3	Collection of NOT-USEFUL click	None	Linear scale	Survey

Table 4.1: It shows the methods that will be used to collect data.

4.1.1 Quantitative Data

Quantitative data was gathered by counting the number of robot comments published during the First and Second Deploy and how many times users clicked NOT-USEFUL.

4.1.2 Qualitative Data

Originally, it was planned to use interviews and a survey for the First Deploy and the Second Deploy. The survey for the First Deploy was skipped due to lack of time.

The First Deploy is to gather baseline data for a new analyzer that provides fix suggestions but presented using initial robot comments in MEAN. After that, MEAN was developed based on the users' feedback. While the Second Deploy is to evaluate the improvements made to MEAN.

4.2 Phase preparation

Understanding how MEAN worked was essential for our work, and would allow us to more easily work with the system in the future. A lot of this understanding was reached through the reading of the previous master's thesis report on the system, as well as fielding questions with people involved in the creation of MEAN, primarily Emma Söderberg, David Åkerman and Jon Sten.

Additionally, considering that the work was performed at Axis, we also needed to understand the infrastructure and workflow at Axis, not only for how MEAN is connected to this infrastructure, but also because we needed to adopt this workflow while working on this thesis. Our introduction to this workflow and infrastructure consisted of common company introduction guides and documents, as well as hands-on experimenting and asking for assistance when these guides and documents weren't easily applied.

4.2.1 Local Setup

In order to ensure that we understood the system correctly, as well as to have a version of MEAN we could freely change, 2 local setups were created, one for each author. These setups took 2-3 weeks to complete. These setups consisted of local Gerrit, Jenkins and RabbitMQ instances running in Docker containers, as well as a local setup of the MEAN project. Local setup of MEAN was done through the help of a setup guide provided on the MEAN open-source repository [24], authored by Mattias Leifsson and Michael Pater. During the setup some issues were encountered when configuring the work environment, which fortunately led to only a small delay in the planned schedule.

4.3 Introducing New Features/Analyzers

With a working local setup, and MEAN having been put into operation and working properly, the next step was to start introducing new features to MEAN, the main one being fix suggestions. MEAN only relays the information provided by analyzers, thus finding a analyzer that provides fix suggestions was key. After looking through the list of already integrated analyzers and finding that none of them supported fix suggestions, the decision was made to incorporate new analyzers with these capabilities.

The list of analyzers, that the authors considered, and that supports fix suggestions, was:

- Clang Tidy
- Black
- ErrorProne
- Auto Python Code Suggestions
- Frama-C
- Cpplint for C++

After asking two teams, related to the Axis supervisors, what analyzers they would find useful, Black, a Python formatting analyzer, was chosen as the first analyzer to be integrated, and was integrated in the First Deploy. Black, being a python code formatter, by default operates directly on code, automatically reformatting the code. However, by enabling an alternative operating procedure with the command "–diff", Black will instead suggest the reformatting it wants to perform. These suggestions can then be treated as code suggestions.

However the authors of this thesis wanted to utilize an analyzer that required the use of dependencies, as so far all analyzers implemented in MEAN analyzes each file individually. The reason being that the authors of this thesis wanted to test how well the integration of an analyzer requiring dependencies could be managed. After further discussions with the supervisors, Error prone, a Java analyzer, was chosen as the second and last analyzer to be integrated, and was to be integrated in the Second Deploy. Error-prone was later discarded in favor of Clang-tidy, a C++ analyzer. Error prone was skipped due to Java not being used that much in the team at Axis.

MEAN was deployed at Axis for one team for the First Deploy and for two teams for the Second Deploy. The teams will be referred to as Team A and Team B, where Team A uses a lot of Python and Team B uses a lot of C++.

4.4 First Deploy

To carry out the First Deploy, some changes have been made to the MEAN system code to be able to run MEAN on the axis servers instead of locally, for example using Axis instances of Jenkins and Gerrit instead of local instances. In the First Deploy of MEAN, support for the Black analyzer and filter module was included. The filter module filters robot comments which originates from previous commits. The fix suggestion in the First Deploy was implemented as a part of the description in the robot comment and the analyzer’s name was added in the category field (see Figure 4.2).

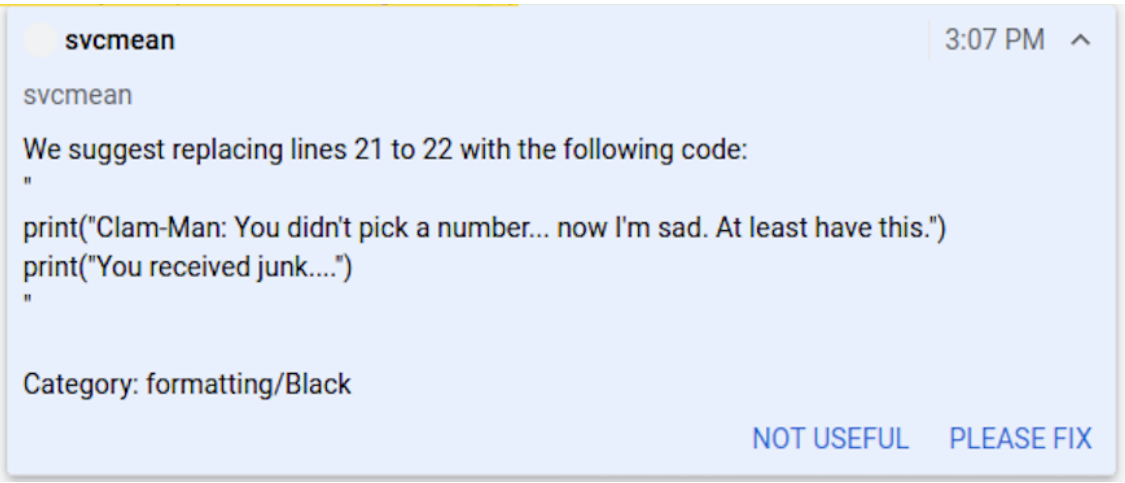


Figure 4.2: A robot comment from Black analyzer from the First Deploy where the fix suggestion is a part of the description.

4.4.1 Running MEAN On The Axis Servers

Up until the First Deploy, MEAN was always run locally. However, in order to run MEAN on any Axis project with a more powerful machine than the ones used by the authors, it would be necessary to move the MEAN files and Docker containers to an Axis server and make the correct Jenkins, RabbitMQ and Gerrit configurations to enable MEAN to run. A bonus with using the Axis RabbitMQ service was that messages sent to the different queues used by MEAN were automatically logged on the axis servers, allowing for easy access to messages sent to RabbitMQ.

4.4.2 Black

To be able to run black in MEAN, a wrapper layer had to be implemented. The wrapper layer handles communication between MEAN and Black. The implementation was done using Docker containers.

Implementation of Black

The steps below describes the wrapper of the Black analyzer:

1. Retrieve all files to be reviewed from the MEAN analyzer request.

2. Loop over the files and run the command:

```
black --diff <source>
```

3. Collect and transform the result from Black and send it as a note to the MEAN robot publisher.

The output from Black comes in the form of a set of regions. Each region consists of a header and lines to remove and add in that region. The header is denoted by "@@" followed by relational information where in the file it applies. Lines that are to be removed are indicated by a minus sign and lines that should be added are indicated by a plus sign.

The note that will be sent to the MEAN robot publisher is in the form of a JSON-object which includes the description, category, range and (path of the file) for the robot comment.

Thereafter the output from Black was transformed into notes which are sent to MEAN as robot comments. The output from Black was split into multiple regions. If Black return more than one region, treat these separately. A region is one or more lines (starting with a minus sign) listed together or one or more lines (starting with a plus sign) listed together (see Listing 4.1).

The Black wrapper was first implemented by reading the output from Black line by line, adding each row starting with a minus sign to a list and the same for each row starting with a plus sign and then to match those list together. This strategy for the wrapper ended up having issues, e.g., the output of Black showing that it will change two rows for one row or vice versa. The implementation was later updated to handle code region by region through the use of regular expressions. This allowed the system to handle a wider variety of results from the Black analyzer.

Listing 4.1: An example of result by using command line "Black diff"

```
@@ -1,6 +1,8 @@
 import random
+
+
 def axisMan():
     print("So you found me... would you like a camera? =D")
     x = ""
     R = ""
     B = ""
@@ -9,11 +11,11 @@

 def clamMan():
-    print("Clam-Man: Hi! I am a Clam-Man")
     name = "Clam-Man"
     age = "???"
```

```
-     address = 'None of your business'
-     pearlie = random.randint(1,3)
+     address = "None of your business"
+     pearlie = random.randint(1, 3)


  if __name__ == "__main__":
      clamMan()
```

4.4.3 Filter Module

The filter module was a recent addition to MEAN made by masters thesis students Mattias Leifsson and Michael Pater at Bosch [9]. This addition allows the system to be more specific in which comments to send, by filtering out comments related to lines of code not changed in the current commit and as such only sends comments for code that is actively being modified by the developer. This addition makes working with MEAN a lot easier since the errors shown have a much higher probability of being useful to the developer.

Inclusion of this module required a small change in a Jenkinsfile, and verification using the local setup to ensure that including the addition did not lead to unforeseen side-effects. With this, the module could be included in the First Deploy.

4.4.4 Pilot Study

Several pilot studies were conducted before deployment in order to ensure that the MEAN system and fix suggestion messages from Black were well presented and easy to understand. These pilot studies were carried out with supervisors where pictures of robot comments were evaluated and compared. An example of a change made by the pilot study was to add analyzers' names in the category instead of just having the error type (see Figure 4.4 in Section 4.5.1). This change helps to know which analyzers generated these fix suggestions.

To ensure that the interview protocol would work, several internal pilot studies were conducted to test the interview protocol. One of the pilot studies was that the supervisors reviewed the language and the structure of the interview protocol. There were also two interview tests. During these interviews one of the authors guided the participant through the interview scenario. Meanwhile, the other author observed the interview and took notes of improvements and clarifications that could be done to the interview protocol. At the same time, the interview time was noted in order to get an estimate of how long time a

regular interview would take. The participants were software developers with previous experience of MEAN.

4.4.5 Interviews and Observations

Ahead of the interviews, a consent paper was sent, it described the goals and terms of the interview, clarified to the interviewees that they could decline to answer any question at any time, and the interviewees where also asked for permission regarding how data was gathered, such as checking to see if the interviewee consents to sound and screen recording during the interview.

The interview (protocol) was sectioned into four parts:

Background: The first is about participants' backgrounds where open and closed questions were used.

Scenario: The participant tests MEAN by following a scenario which was divided into tasks where one of the authors presented the task to participant. Meanwhile, if the participant had declined sound and screen recording, then the second author observed and took note of the participants actions. Otherwise observations were done after the interview by watching the recorded material. The observation looked for details that might stand out, for example if the participant did something wrong, succeeded with the task, etc.

Questions post scenario: The third part contained questions about participants' opinion of MEAN for the tasks he/she did. Some of the questions were also specific to the new analyzer added since the First Deploy, e.g. whether the fix suggestions provided were easy or bad, were useful, etc.

MEAN score: Finally, several questions were asked in the form of linear scale where the participant would score for the entire MEAN system.

4.4.6 Collection of Feedback

The thesis authors have analyzed the feedback received from the interviews where the open-ended questions have been collected and checked what most participants thought. Some of the questions asked the participants to score aspects of MEAN on a scale from 1 to 5, which we have documented in a diagram (more details in chapter 5).

After the end of the First Deploy, the following data was collected: how many NOT-USEFUL clicks made, robot comments (for each analyzer) published, and

the number of changes and developers that interacted with MEAN. Additionally, the authors have contacted those who clicked on NOT-USEFUL in order to know why the button was clicked.

4.5 The Second Deploy

Proper fix suggestions have now been implemented, allowing for a clear comparison between fixed and unfixed code as well as automatic application of fixes. In order to test fix suggestion on a team other than Tools at Axis, support for a new analyzer, Clang-tidy, was added. It can be used by teams that produces C++ code. With the choice of Clang-tidy it was possible to see how developers interacted with an analyzer that provide fix suggestions and not just formatting suggestions as Black does. MEAN was also improved to include information links in the robot comments published to Gerrit. Also, for the Black analyzer a bug was fixed and the wrapper was developed to give specific messages for the addition or removal of lines. Black continued to be used by the previous team in the First Deploy.

During the Second Deploy, some statistics/comments were lost due to technical issues. These had to be recovered using alternate means.

4.5.1 Clang-tidy

The static code analyzer Clang-tidy was an early candidate for adoption into MEAN, as there was evidence for an implementation being possible, since it is integrated into Tricium [6]. But, as mentioned in Section 4.3, the authors wanted to implement two static analyzers. The static code analyzer Black was already used in the Team A the thesis authors had contact with naturally made it a priority over Clang-tidy, leading to Clang-tidy not being chosen for adoption into MEAN.

However as the project carried on, Error Prone turned out to be harder to integrate than initially thought and less useful in Axis developer teams beyond Team A than was expected. As such, seeing as having at least two analyzers to show a general implementation of fixes was preferable, after a period of discussion within the team Clang-tidy was chosen to be the second analyzer. The Clang-tidy analyzer was also a good match for Team B which develops in C++.

Implementation of Clang-tidy

A Docker container has been created for Clang-tidy that contains all software and scripts to run clang-tidy, such as LLVM, Clang-tidy, Python, etc. A wrapper script was written in Python and performs the following for each file in the change:

1. Run Clang-tidy and get fix suggestions in the form of a 'yaml' file using command line:

   ```
   clang-tidy --export-fixes=<filename> -checks=<string> <source0>
   ```

   Clang-tidy was instructed to run all checks by providing -checks=*

2. Convert 'yaml' to 'json' file

3. Interpret the results of the file and send it as a note to MEAN robot publisher.

The output from Clang-tidy needs further processing before it can be sent to Gerrit. An example output can be seen in listing 4.2. The start position needs to be converted from byte offset from start of file, to number of lines from start of file.This is done by counting the number of line feeds "\n" between start of file and reported offset.

In some situations, since all checks are enabled, Clang-tidy will report multiple warnings for the same location and problem but with different category(see Figure 4.3). The solution to this problem is to report categories in a single robot comment by grouping all warnings that have the same line number and description. (see Figure 4.4). Another issue was that, if multiple checks provided the same suggestion, then no replacement was provided. This issue was resolved by collecting all check that reported the same suggestion and rerunning clang-tidy with this reduced set of checks. The result is that clang-tidy reports the same diagnostics, but this time with replacements.

Listing 4.2: Example of how JSON file look like by using Clang-tidy command line "–export-fixes". DiagnosticName represents the type of "check" and "Message" is the description of fix suggestion. FileOffset and Offset represents location of fix suggestion in the code.

```
{
"MainSourceFile": "test.cpp",
 "Diagnostics": [
    {
    "DiagnosticName": "readability-container-size-empty",
    "DiagnosticMessage":{
```

```
        "Message": "the empty method should be used to check for
           emptiness instead of comparing to an empty object",
        "FilePath": "test.cpp",
        "FileOffset": 68,
        "Replacements" : [
            {
            "FilePath": "test.cpp",
            "Offset": 68,
            "Length": 8,
            "Replacement": "text.empty()",
            },
            ]
        },
    ]
}
```

```
6 warnings generated.
/app/test.cpp:5:18: warning: prefer using 'override' or (rarely) 'final' instead
 of 'virtual' [cppcoreguidelines-explicit-virtual-functions]
    virtual void reimplementMe(int a) {}
                 ^
note: this fix will not be applied because it overlaps with another fix
/app/test.cpp:5:18: warning: prefer using 'override' or (rarely) 'final' instead
 of 'virtual' [hicpp-use-override]
note: this fix will not be applied because it overlaps with another fix
/app/test.cpp:5:18: warning: prefer using 'override' or (rarely) 'final' instead
 of 'virtual' [modernize-use-override]
note: this fix will not be applied because it overlaps with another fix
/app/test.cpp:8:18: warning: prefer using 'override' or (rarely) 'final' instead
 of 'virtual' [cppcoreguidelines-explicit-virtual-functions]
    virtual void reimplementMe(int a) {}
                 ^
note: this fix will not be applied because it overlaps with another fix
/app/test.cpp:8:18: warning: prefer using 'override' or (rarely) 'final' instead
 of 'virtual' [hicpp-use-override]
note: this fix will not be applied because it overlaps with another fix
/app/test.cpp:8:18: warning: prefer using 'override' or (rarely) 'final' instead
 of 'virtual' [modernize-use-override]
note: this fix will not be applied because it overlaps with another fix
```

Figure 4.3: Example of clang-tidy result when using all checks.

Figure 4.4: Example of clang-tidy robot comment in Gerrit, this image has been taken before the implementation of fix suggestions and link info.

4.5.2 Fix Suggestions Info into Gerrit

By providing Gerrit with fix suggestions, Gerrit will automatically be able to apply the fix when requested by the developer. This saves time for the developer since there is no need to manually copy the fix from the robot comment, apply it locally and upload a new patch set. A SHOW-FIX button will be shown whenever a robot comment contains a suggested fix. When that button is pressed, a window will be opened which contains the code with and without suggestion and with the APPLY-FIX button which makes it possible to save a suggestion (see Figures 4.5).

Implementing of Fix Suggestions Info

Gerrit has a REST-API for creating robot comments. These comments are created using a JSON object called RobotCommentInput, which is data that is sent using a POST request to the REST-endpoint. An optional argument for Robot-

CommentInput is the fix suggestions field, which suggests corrections for this robot comment as a list of FixSuggestionInfo. FixSuggestionInfo describes change in the file what it should look like and consists of (fix id, description and FixReplacementInfo). FixReplacementInfo is a list that will describe how one or more files should be changed based on the suggestions and it consists of (path, range and replacement) (see Listing 4.3).

Listing 4.3: An example of how fix suggestion data looks like

```
FixSuggestionInfo = {
   "fix_id": ,
   "description": ,
   "FixReplacementInfo":[
      {
         "path": ,
         "range":{
            "start_line": ,
            "start_character": ,
            "end_line": ,
            "end_character": ,
         },
         "replacement": ,
      },
   ]
}
```

Figure 4.5: A robot comment from Clang-tidy analyzer where the check has fixes.

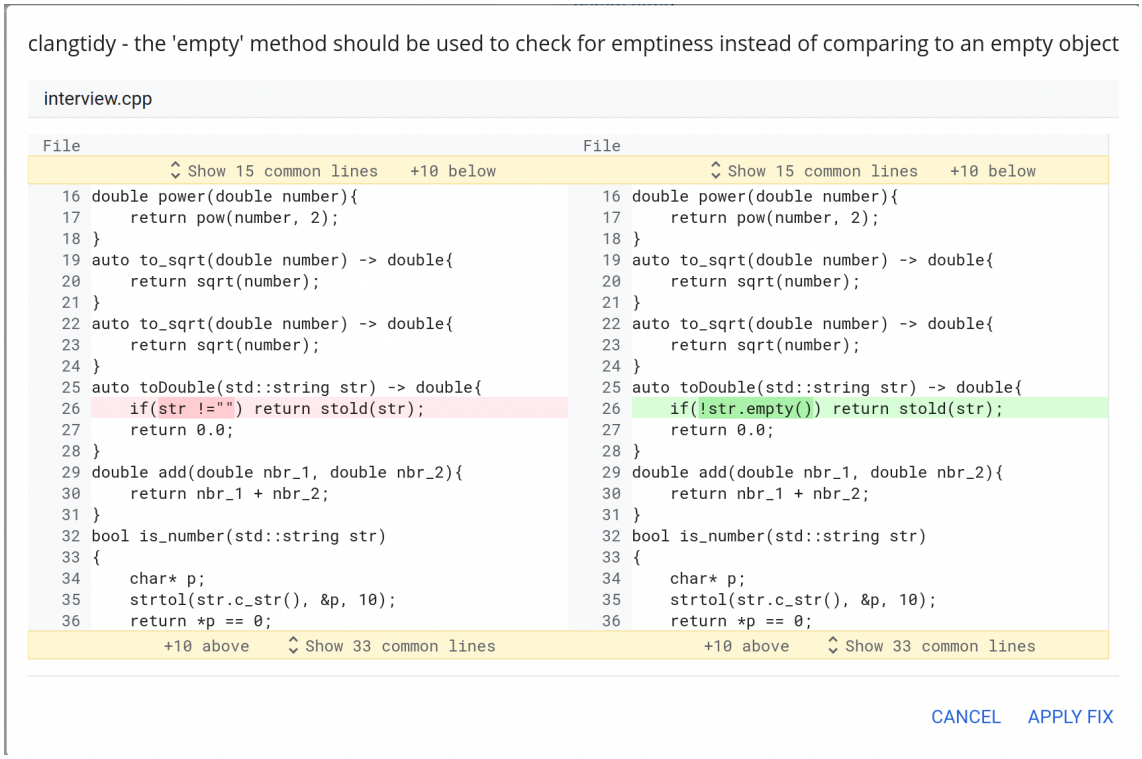


Figure 4.6: Overview of how the SHOW-FIX window looks like.

4.5.3 Link Info into Robot Comment

Gerrit API in RobotCommentInput has an optional argument for link information. When provided it shows a "Run details" button in the robot comment. When clicked, it opens the provided link. This argument is limited to a single link. Unfortunately that causes problems with robot comment reported by MEAN, since a single comment may contain multiple categories of problems, and thus multiple links. The decision was made to instead put the links directly in the message part of the comment (see Figure 4.5).

4.5.4 Pilot Study

It was not necessary to conduct a pilot interview for the Second Deploy since there were only minor changes to the interview protocol and the authors were more experienced in interviewing. Instead, a pilot study was made in order to check if the implementation of new features worked as it should and that the appearance of the robot comments and fix suggestions are clear and easy to follow. For example, it was verified that the problems reported by Clang-tidy was displayed on the correct line and had the correct fix suggestion, etc.

4.5.5 Interviews and Observations

Changes were made to the interview protocol, and in order to increase the number of developers providing feedback, a new team was added for the Second Deploy. This team was chosen among other things with the ability to test Clang-tidy in mind, meaning they needed to use Clang-tidy compatible code, as the team used in the First Deploy did not produce a sufficient amount of C++ codes. This lead to the Second Deploy having two teams, the old team which is mostly used to test Black and writes Python, and the new team which is used to test Clang-tidy using C++.

Scenario

In the scenario, a sub-task was added to task number 5 (see interview protocol in Appendix A). Each participant was given the task of fixing at least three robot comments, but was not instructed how to do so. A sub-task was added so that if the participant decided to fix the comment by hand, whether locally or within Gerrit, then the participant was instructed to try again, but this time using the APPLY-FIX/SHOW-FIX functionality. The reason for this was to see how participants would react to these new features.

4.5.6 Survey

In order to get feedback from the developers who were exposed to MEAN but were not interviewed, a survey was created. The goal with the survey was to reach a larger target group. It was sent to all the affected developers (owners, reviewers, developers on CC, listeners and submitters) a week before the end of the deployment.

Survey has been divided into three parts (see Appendix B):

- To see which team got the most feedback, a question was asked regarding which team at Axis the developer was currently working in. After that the developer was given a reminder of what the robot comments and suggestions from Black and Clang-tidy analyzer looked like.

- The second part had questions about which analyzer the user got results from. A series of linear scale questions were asked where the survey respondent states their reflection on MEAN. For example; did they believe the suggestions were well implemented and were they useful? Another important question that was asked was whether the respondent would, would not, or would maybe appreciate having MEAN running on Axis repositories in the future. The respondent was also asked to motivate why.

- To see which parts of MEAN the developers at Axis would be interested in, if MEAN was made available in the future at Axis. A question was asked about how much they will use features of MEAN, for example NOT-USEFUL, PLEASE-FIX, SHOW-FIX etc. buttons. Thereafter if the user has ideas for new feature or improvements for existing ones in MEAN.

The survey was sent via email to the developers (owners, reviewers, developers on CC, listeners and submitters) who interacted with MEAN during the Second Deploy. To get these developers, a python script was implemented, which gathered the affected developers of Gerrit changes using the Gerrit API. Using this API it also got the number of projects that received robot comments from MEAN. From this list of projects it was possible to retrieve changes that were created and modified during the Second Deploy.

4.6 Data Analysis

This section describes the data analysis methodologies for both First and the Second Deploy.

After the deployment, a lot of data had been gathered in several forms:

- Interview recordings
- Interview questionnaire answers
- Gerrit statistics

This data had to then be collected and analyzed. While the collection of data was performed continuously during the deploy, a more in-depth analysis of the data was performed after the deployment.

For the recordings, analysis was done through the use of observations. These observations did not follow a set structure, instead emphasis was placed on finding specific or unanticipated behaviour, as well as documenting if the subjects shared some sentiments regarding the scenario. One review subject requested that no recording be made, necessitating the authors to rely on notes taken during the interview.

The questionnaire answers are used to help deduce the behaviour during the interviews as well as provide more insight into the sentiments the subject has towards MEAN. It is important to emphasize that these results will be considered both individually and jointly. There might be some cases where the observation and the subjects account differ, which would without this approach be overlooked.

As for the analysis on the Gerrit statistics, there were mainly three statistics that were looked at:

- Number of NOT-USEFUL clicks (in total and for each analyzer)
- Number of comments (in total and for each analyzer)
- Number of changes affected

These are used to provide a picture as to how well-received the comments from the analyzers were by comparing NOT-USEFUL clicks to the total number of comments. The number of changes were included to give a picture of how widely MEAN was actually used.

Data analysis for the Second Deploy is the same as done in the First Deploy except that periodic checks were made to see if someone had reported a comment as NOT-USEFUL. The reason for checking periodically was to get feedback as soon as possible and find bugs that could affect the investigation. Bugs which the authors could then fix.

In order to retrieve some comments that were not properly saved, Gerrit statistics have been taken from the Gerrit API by sending a request to Gerrit to get the robotcomments.

Chapter 5

Results

This chapter will report the results found during the evaluation phases of the First Deploy and the Second Deploy, further described in Chapter 4. MEAN was deployed for 6 weeks at Axis where 2 weeks were deployed for the First Deploy in the Team A and 4 weeks for the Second Deploy in the Team A and Team B.

During the First Deploy and the Second Deploy, 35 projects were affected via 257 changes leading to interaction with MEAN from 66 developers (owners, reviewers, developers on CC, listeners and submitters) (see Table 5.1). Among the 35 affected projects, 4 were projects used by the thesis authors for testing the improvements of MEAN.

	Amount
Projects	35
Changes	257
Developers	66

Table 5.1: The total amount of projects, changes and developers interacting with MEAN during the First Deploy and the Second Deploy.

5.1 Results From the First Deploy

This section reports the results of the interviews and Gerrit statistics carried out as part of the First Deploy. During the First Deploy, 9 projects were

affected via 88 changes leading to interaction with MEAN from 14 developers (see Table 5.2).

	Amount
Projects	9
Changes	88
Developers	14

Table 5.2: The total amount of projects, changes and developers interact with MEAN during the First Deploy.

5.1.1 Interviews

This section will include the results obtained from interviews conducted with 3 people at Axis. The interview participants will be referred to as P1, P2 and P3. Note that the interviews in the First Deploy were performed in Swedish therefore all quotes has been translated to English.

The participants had experience with software development and they have worked with program analysis indirectly (i.e. they use program analysis tools before the code is pushed up for review but they have not worked with development of these program analysis tools). The majority of participants use program analysis on a daily basis, especially Black which is mentioned by participants (P1, P2 and P3).

The general attitude from developers towards using MEAN in their workflow was positive.
An example of a positive quote:

> "*I like when you automatically get notifications, which you then can go and fix.*" (P2)

Some participants were positive about using MEAN but negative about using Black:

> "*Its good that it [MEAN] highlights potential problems, but that is not useful for Black because you can run black locally and apply all fixes automatically.*" (P3)

Some participants said that the comments are easy to understand and some said that they are not immediately easy:

> "*I was a little confused at first, as the reason for a change is not given*" (P1)

> "*Yes, it's easy to understand the description of the comments*" (P3)

Participants mentioned some things they would like to be improved:

> "*Provide information links for more details.*" (P1)

> "*It would be better if the error was immediately provided, without unnecessary extra text [We suggest... (see Figure 4.2 in chapter 4)].*" (P2)

> "*The name of the service-account [in MEAN] is not needed twice*" (P3)

Some participants mentioned that enabling Black to provide fix suggestions would be helpful and others not.

> "*Yes, It is better to let the developer decide what to change in the code*" (P2)

> "*Running analyzers locally is how I usually do things, especially in the case of Black which is opinionated. There is no reason to consider whether a result from Black should be used or not. It is also quicker to fix all errors in Black at once locally than fixing them one by one [in Gerrit].*" (P3)

At the end of the interview, participants were asked to grade properties of MEAN on a Linear scale (1-5). The results of this part of the interviews is shown in Figure 5.1, showing that the comments are well presented and that the suggestions are useful. Furthermore, suggestions are also reasonably presented and it is quite easy to interact with comments.

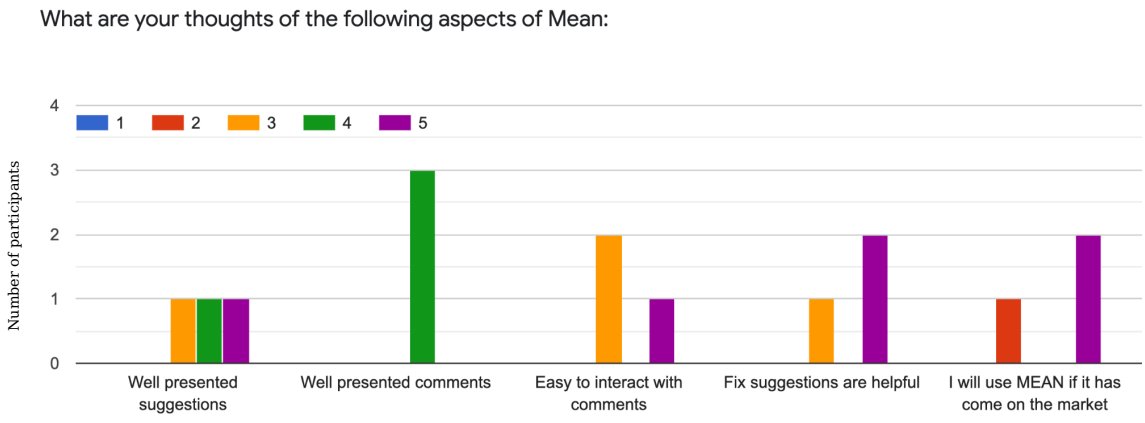


Figure 5.1: Results of participants' opinion of MEAN in the First Deploy.

The reason for one of the participants lower grading regarding the last question in Figure 5.1 was:

> "*If MEAN is to be released in this version, then my score is 2, while if I consider the potential of the system, such as more analyzers available and less false positives, then the score is 5.*" (P1)

5.1.2 Observations

This section reports the participants' reactions, ideas, activities, erroneous actions and wishes from the scenario part of the interviews, where the authors have watched these recorded interviews together and analyzed them. During each interview, as mentioned in Section 4.4.5, each participants actions and answers were either recorded on video or by hand for later analysis of their interactions with MEAN. The tasks to be performed were summarily to solve a number of errors in a code sample created by the developers, using MEAN robot comments as a guideline. These were our findings:

- All participants had experience with usage of Black from using it in their workflow.
- P1 ran Black directly on code and the authors of the thesis had not prepared for this happening, as the objective of the test was to observe how well the robot comments worked with finding and fixing errors. In later interviews, it was mentioned that using Black locally is not allowed, indicating that his workflow with Black does not fit well with this new approach.
- P1 missed to fix comments he had pressed PLEASE-FIX on
- P1, P2 and P3 misunderstood the purpose of PLEASE-FIX, they thought that it would allow them to automatically fix the warning.
- The experience of using Gerrit varied among the participants, with two participants being very familiar and one having less experience from only using it for a couple of months.
- A couple of mistakes were made by P2, including a faulty fix and a missed fix. For example, P2 fixed a suggestion in the wrong place even though the robot comments contained the start and end line of the change.

5.1.3 Published Robot Comments

During the First Deploy, MEAN produced a total of 109 robot comments and got 61 NOT-USEFUL in total. Black, Hadolint and Ansible-lint generated 97, 8, and 3 robot comments respectively, and 60, 1 and 0 comments respectively were reported as NOT-USEFUL, which corresponds to 98 %, 2 %, and 0 % of total NOT-USEFUL clicks (see Table 5.3).

Analyzers	Robot comments	Total NOT-USEFUL	$\frac{Total\ NOT-USEFUL}{All\ NOT-USEFUL}$
Black	97	60	98 %
Hadolint	8	1	2 %
Ansible-lint	4	0	0 %
All	109	61	100 %

Table 5.3: The total amount of robot comments and NOT-USEFUL clicks for the different analyzers running in the First Deploy. The percentage is rounded to an integer number.

All NOT-USEFUL clicks on robot comments from the Black analyzer were made by one developer. The reason was that Black did not take care of blank lines in the First Deploy so therefore Black had a bug that reported suggestions on the wrong line.

> *"I pressed "NOT USEFUL" to code suggestions I received in the comments made no sense, for example"*
>
> ```
> def __repr__(self) -> str:
> ```
>
> *"and the robot comment:*
> *Please consider replacing line 393 with the following code: "*
>
> ```
> def __init__( self, address: Tuple[str, int] =
>     ("", 0), handler: Any = FTPHandlerDummy ) -> None:
> ```
>
> *"So here I clicked NOT-USEFUL"* (User in the First Deploy)

5.2 Results from the Second Deploy

This section reports the results of the interviews, Gerrit statistics and survey carried out as part of the Second Deploy.

During the Second Deploy, 31 projects were affected via 169 changes leading to interaction with MEAN from 56 developers (see Table 5.4).

	Amount
Projects	31
Changes	169
Developers	56

Table 5.4: The total amount of projects, changes and developers (owners, reviewers, CC, listeners and submitters) that interacted with MEAN during the Second Deploy.

5.2.1 Interviews

The interviews were performed with 4 participants, 2 from Team A and 2 from Team B. With slightly different MEAN configurations for the users of MEAN in the Second Deploy. The results will be presented for users of Black and then for users of Clang-tidy. The interview participants will be referred to as P1, P2, P3 and P4. The interviews in the Second Deploy was performed in Swedish therefore all quotes were translated to English.

Black

One participant wanted to see MEAN as part of their normal workflow, while the other considered MEAN more as a way to catch bugs if there are bugs the developer hadn't already checked for:

> "*To make these apply fixes and let MEAN fix the problems in Gerrit, is something I could consider if I felt I was completely done with the change. And to just fix minor errors... I can see a certain amount of usefulness in that, clearly. However I would rather have seen that everyone working with code that is easy to lint, gets it into their workflow [to run the analyzers]. So that MEAN always in actuality runs in vain.*" (P4)

A negative was mentioned regarding the interaction with comments, specifically the lack of information in Black descriptions:

> "*...here [in this comment] it only says "Yes, the formatting is incorrect." But in this case Black could me that "Don't use single quotes.""*" (P4)

Results showed that suggestions are well implemented and it is easy to integrate with comments while one participant considered comments to be well presented while others considered comments to be very well presented. Thus fix suggestion is useful and fix application is quite well received (see Figure 5.2).

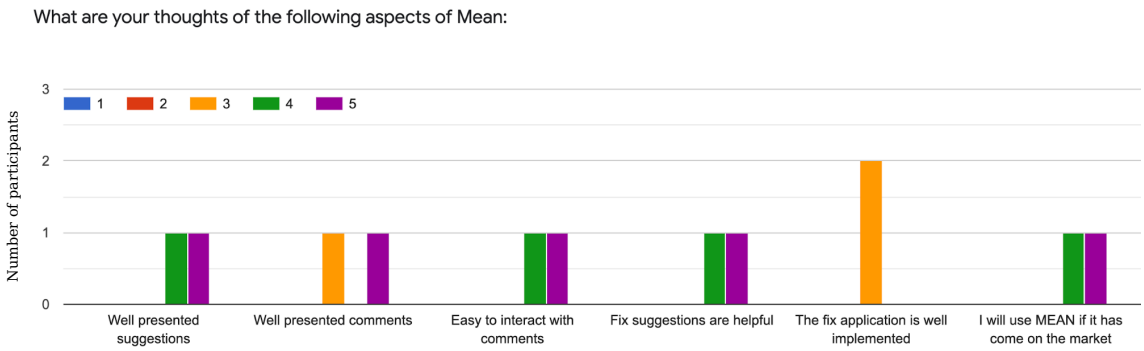


Figure 5.2: Results of participants' opinion of MEAN where participants used Black.

The reason why one participant considered the comments to be not quite well presented was that Black does not provide a motivation for why the suggestion is better.

> "*Description in robot comment is that there is formatting error according to Black analyzer and I do not know why it is better to change with this provided suggestion*" (P3)

Also the reason why they thought that fix application is quite well presented is that the fix suggestion window in Black shows what the line will look like, instead of showing what should change on a line in text.

> "*Black does not clearly show what the difference is, in fact shows what the line should look like. Show the difference, not just the compensation.*" (P4)

Clang-tidy

One participant was interested in using MEAN as part of his regular workflow, replacing their current workflow, while the other participant just wanted this to be an addition to their already existing workflow.

> "*I want to add, actually I do not see a difference MEAN with Jenkins tools that do as we already use*" (P1)

Figure 5.3 suggests that suggestions and comments are well presented and the comments are easy to interact with. While users' opinions about whether fix suggestion are helpful and fix application is well implemented are divided into two parts, with one part being very satisfied. On the other hand, the other participant is unsatisfied with fix suggestion and expressed a neutral opinion on the implementation of fix applications.

> "*I do not usually follow some of the checks that are used and at the same time many of checks give false positive. So I score 2 for false positive checks and 4 without false positive.*" (P2)

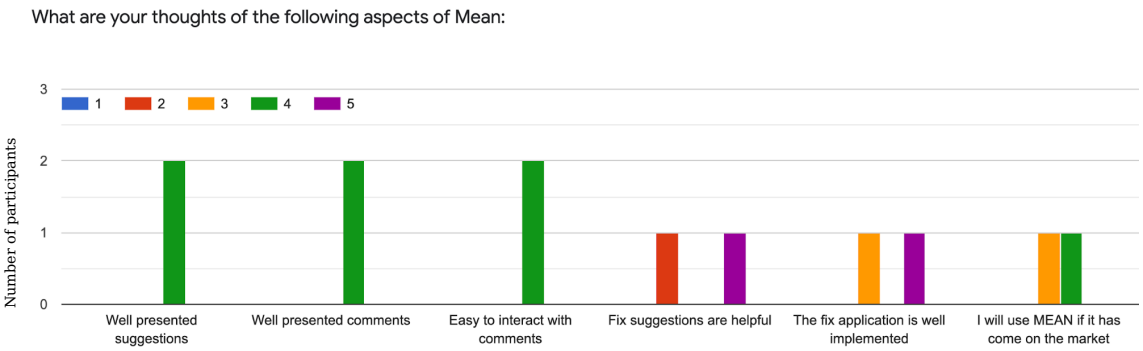


Figure 5.3: Results of participants' opinion of MEAN where participants used Clang-tidy.

5.2.2 Observations

This section reports the participants' reactions, ideas, activities, erroneous actions and wishes from the scenario part of the interviews, where the authors have watched these recorded interviews together and analyzed them.

Workflow differences

- Generally
  - P1, P2 and P4 fixed code locally instead of using the show-fix function, while P3 noticed the SHOW-FIX and APPLY-FIX buttons. Although P4 used SHOW-FIX to show how code will be fixed once the button was pointed out. P1 got a walkthrough of the SHOW-FIX functionality after expressing uncertainty about what to do.
  - P1 used his own comments, rather than the MEAN comments, to fix issues. He also downloaded a copy of the review.
  - P4 mentions that automatic analyzers should be used automatically
- Erroneous actions
  - P3 misunderstood PLEASE-FIX, expressing that he thought it would fix code when pressed.
  - P2 solved a suggestion incorrectly and which led to a compiler's error in the code. He also forgot which comments that were fixed.
  - P4 said that Black comments did not say how comments will be fixed. The authors then directed P4 to the SHOW-FIX button.
- APPLY-FIX

  - All participants have complained about the fact that when pressing APPLY-FIX the user jumps to the change overview screen and ends up in edit mode, which they didn't like. One reason being that they could not fix several of the suggestions at the same time before changing windows. This also made entering the full view diff harder, which P4 mentioned.
  - P4 mentions that it is not clear whether several fixes can be applied at once.

- Description

  - P1 and P2 used the link which is provided in robot comments to know how the code will be fixed, while P3 never used the link information.
  - P4 complained that Black does not give a reason for the suggestion.

- Suggestions

  - P3, P4 found it confusing that comments remain after the application of fixes.
  - P4 complained that fixes in Black show how the line should be instead of saying what should change in the line.

Wishlist

- P3 mentioned new features, including APPLY-ALL button and manual checks on edits.
- P3 mentioned that he would like immediate access to both the diff-view from SHOW-FIX and APPLY-FIX button.
- P4 mentioned that he would have the categories' text with the hyperlink instead of writing the link below the message.

Additional information

- P4 found a bug when adding lines in Black.
- Both P3 and P4 got to see Clang-tidy comments to get an example of more detailed comments.
- P2 mentions that style errors are less important to get errors for than for functional errors.

5.2.3 Published Robot Comments

During the Second Deploy, MEAN produced a total of 1268 robot comments and 98 of these got NOT-USEFUL clicks NOT-USEFUL which corresponds to 8 %. Clang-tidy, Black, Hadolint, Ansible-lint produced 968, 201, 74 and 25 robot comments respectively, and 79, 6, 10 and 3 comments respectively have been reported as NOT-USEFUL, which corresponds to 81 %, 6 %, 10 % and 3 % of total NOT-USEFUL clicks (see Table 5.5).

Analyzers	Robot comments	Total NOT-USEFUL	$\frac{Total\ NOT-USEFUL}{All\ NOT-USEFUL}$
Clang-tidy	968	79	81 %
Black	201	6	6 %
Hadolint	74	10	10 %
Ansible-lint	25	3	3 %
All	1268	98	100 %

Table 5.5: The total amount of robot comments and NOT-USEFUL clicks for the different analyzers during the Second Deploy. The percentage is rounded to an integer number.

Most of the NOT-USEFUL clicks from the Clang-tidy analyzer were due to false-positives that some of the checks have. Some answers from users (using Clang-tidy analyzer) concerning why they reported a robot comment as NOT-USEFUL:

> *"I thought the comments were mostly of the character code style rather than having some really functional value, there are certainly other situations where the static code analysis finds "real" problems, and then it is of course useful. As for code standard, I think it is better to start at the other end, i.e. first decide on a code standard, and then configure a tool according to this standard instead of forcing here something that is radically different from the (unspoken) code standard we follows in the current situation."* (User in the Second Deploy)

> *"I think it was in that review, very new code so it's pretty noisy*
> *A bug is that it says that:"*

```
double f_test(int order_nested, int order_full) const {
...
}
```

> *" It can be static which it cannot as it calls for size from its parent class*
> *Then we turned on Clang-tidy/google-readability-todo, which we do not follow."* (User in the Second Deploy)

> *"It interprets for example the declaration of a String as a function"*
>
> `std::string error_msg = "... (FE_DIVBYZERO)\n"`
>
> *"It also complains about, for example, that you do bit operations on signed, but that's how it works (understand that it can not know that), and then you do not want to cast the arguments there (I think anyway). Is there any way to remove the comments from the review that you do not find useful? At the moment, I think it produces a bit too many comments that I do not think add much, and which make it harder to read the reviews"* (User in the Second Deploy)

Some users' response about why robot comments were reported as NOT-USEFUL for the Ansible-lint analyzer:

> *"Ansible-lint complained about "Variables should have spaces before and after: var_name ", but this is only relevant for files that are subject to Jinja-templating, ie. where var_name has a special meaning. In this case, it was not so, but the file was a completely ordinary text file that happened to contain a substring that looked like a Jinja-expression. Of course, it's not MEAN's fault but a bug in Ansible-lint."* (User in the Second Deploy)

> *"They are writing a logstash filter, and the comments for "too long line" created quite a lot of noise as I wanted the reviewer to focus on something else instead I tested to see if the MEAN comments disappeared if you printed not useful. But it was still there so I ignored the rest."* (User in the Second Deploy)

Examples of reasoning given for why NOT-USEFUL was clicked for the Hadolint analyzer:

> *"Those comments are about sticking apt packages in Docker files and it is extremely rare to do so. The reason is that you usually want to get package updates by just rebuilding the image. So in most cases this is a conscious choice and the Hadolint comment is just noise."* (User in the Second Deploy)

One of the users in the Second Deploy clicked NOT-USEFUL 6 times on comments from the Black analyzer:

> *"It [the comments] came long after we uploaded several new patch sets"* (User in the Second Deploy)

The reason behind this feedback was that some components of the MEAN service sometimes shut down unexpectedly leading to messages piling up in the message queues. These were then handled when the service was restarted, leading to a time delay.

5.2.4 User Survey

The survey was sent to the 56 developers that had interacted with MEAN. Thereafter, the survey received 12 responses, 5 from developers working in the Team A and 7 from developers working in the Team B. In this group of developers, 7 received robot comments from Black, 7 from Clang-tidy, 1 from Hadolint and none from Ansible-lint (see Figure 5.4). The survey responses will be referred to as SR1-SR12.

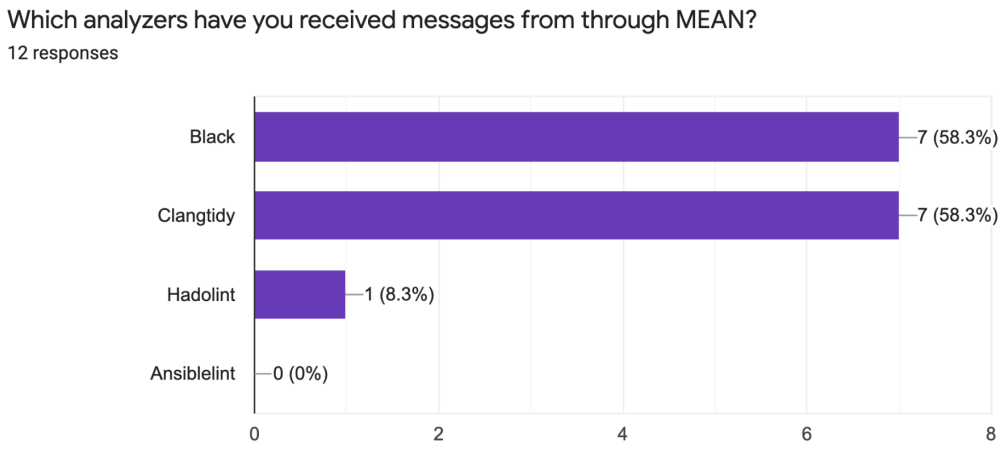


Figure 5.4: Result of which analyzers developers received robot comments from via MEAN.

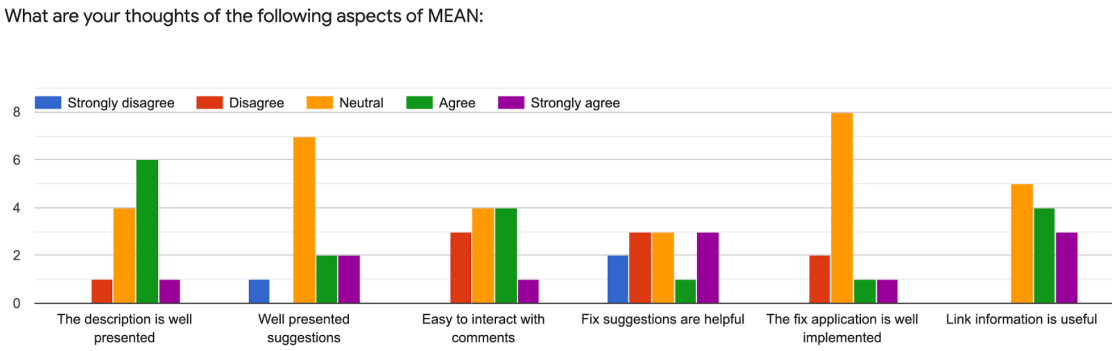


Figure 5.5: Results of participants' opinion of MEAN.

Figure 5.5 shows that the developers' responses of the description of robot comment, the presentation and implementation of suggestions, all lean towards the positive end. There are some positive responses for "Fix suggestions are helpful" and "Easy to interact with comments", and some are negative. While the responses for links lean towards positive, with the lowest rating being neutral.

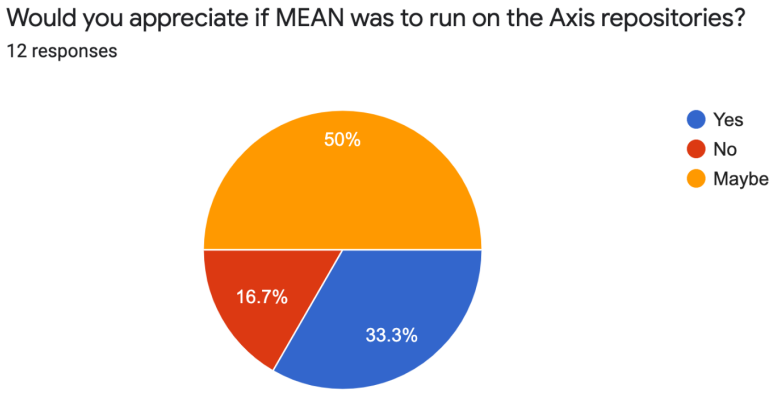


Figure 5.6: Results about whether developers would or would not appreciate to run MEAN on Axis repositories.

Figure 5.6 shows that 50 % of developers answered that they might appreciate if MEAN ran, 33.3 % of developers that they would appreciate if MEAN ran, and 16.7 % answered that they would not appreciate if MEAN ran.

An example of positive responses where a survey user would appreciate if MEAN ran on Axis repositories:

> "*If it was to be unappreciated, I'd assume it would be because the comments take up quite a lot of screen space, making it a little harder to review code. The pros of running MEAN would be: - lessen the need for setting up Jenkins jobs that do linting for every Gerrit project individually, MEAN could take care of that - The fix suggestions feels like a cheap and easy way to get around failures that doesn't require much thought, i.e. linting stuff*" (SR3)

> "*It would enforce a higher code standard broadly on Axis. And that is a good thing.*" (SR6)

> "*MEAN is great. To a smoother UX for previewing and applying suggestions is kinda needed.*" (SR7)

> "*I think static analyzers can be useful. I haven't actually interacted with MEAN so I don't have any experience with it but it seems like it can be useful.*" (SR11)

Example of negative responses where users would not appreciate if MEAN ran:

> "*Way too many false positives*" (SR1)

> *"I have only tried clangtidy, but I find that it has too many false positives and irrelevant or opinionated suggestions, which makes the reviews hard to read for the reviewers. If it was possible to filter out the mean messages in the gerrit UI, this would be less of an issue."* (SR4)

Examples of responses from users who would appreciate if MEAN ran on Axis repositories but they have comments about things that could be improved:

> *"Correctly set up it can be useful. However one generally needs to do so at the beginning of projects to avoid it being to noisy. Many false positives causes one to ignore a lot of output. "* (SR2)

> *"Think the comments and descriptions themselves looked good. Although, the problem is that we already use black with some different formatting rules, such as a longer line length. Therefore, the comments were not really that useful to us."* (SR5)

> *"Anything that MEAN complains about is something that I'd want to have available during local development, i.e. I want all linters to be run via tox, make, Maven, or whatever I'm using. IOW, if MEAN complains about my code it's either because it's running a linter that I'm not aware of, in which case I'll probably add it to the arsenal of linters already run, or because it's a linter I've consciously rejected, or because it's not picking up any local rules that I might have. Of these three cases only the first case gives my valuable feedback. So from my perspective, MEAN brings value when it checks things that I haven't thought about (yet agree with)."* (SR9)

> *"+ Good tool when helpful - LOTS of noise, suggestions that are "opinionated", "this is how google/llvm/fuchsia-code style says how to do it", and some of those are even in conflict - False positives - The "Apply fix" is hard to apply for multiple fixes in the same change (not obvious on how to get back to the diff comments, no tracking on which fixes have been applied). Configuration possibilities must be easy for each repo+branch most likely so that maintenance branches do not get invaded by rules that might be in effect on master. Rules must probably be opt-in instead of optout, just cause of the clang-tidy situation, or maybe only do clang-tidy if a ".clang-tidy" config exists in the repo?"* (SR10)

> *"The "sensitivity" needs to be toned-down, I think the useful comments tend to drown in nit-picking on pure subjective matters. I think it is a bit backwards to include this in the normal review-step. The web-gui is used to simplify code-review-interaction between developers, to include static code analysis at the same time makes the*

normal reviewing-process more difficult to overlook. The comments from the static code analysis is better handled by the individual developer before asking other developers to spend time on reviewing (I have no opinion regarding if this is most efficiently done in a web-gui or pure text-based just like normal compiler-feedback, I'm sure different persons have different preferences in this matter)." (SR12)

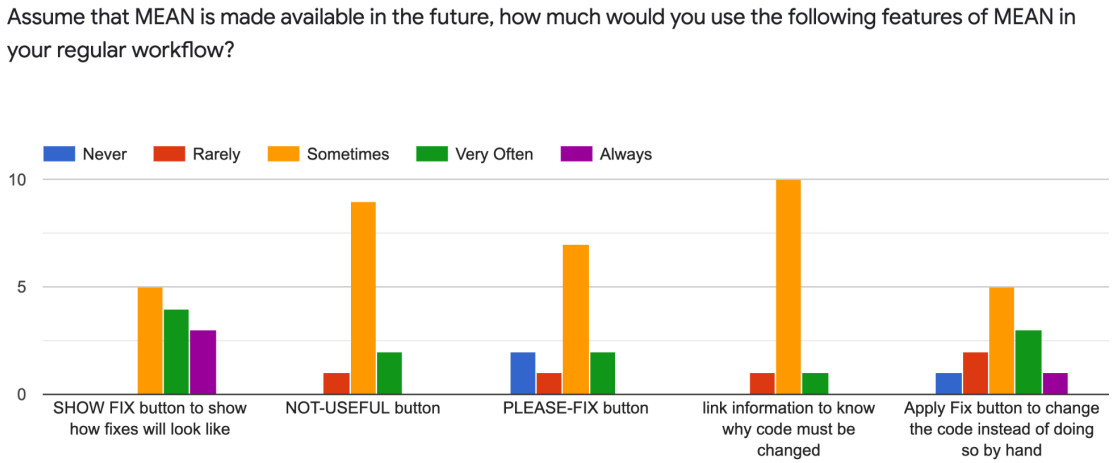


Figure 5.7: A diagram of how likely the participant thinks it is that they'd use certain features of MEAN, if it were to become available in the future.

Figure 5.7 shows that if MEAN is made available at Axis, the respondents report that they would sometimes use the NOT-USEFUL button, the PLEASE-FIX button, and the link information, and some of them will use SHOW-FIX very often while the others sometimes. For the APPLY-FIX button, the response was quite mixed, with some positive and some negative responses alike.

Chapter 6
Discussion

In this chapter will be discussed the results related to the research questions. The interview participants from the First Deploy will be referred to as D1P1 to D1P3 while the participants from the Second Deploy will be referred to as D2P1 to D2P4.

6.1 RQ1: Acting on Fix Suggestion

In the First Deploy, the Black analyzer was implemented to see how developers interact with fix suggestions by robot comments. The interaction with suggestions in the First Deploy was via robot comments where the message and fixes are a part of comments. The users also did not have any link information to read more detail about why fixes should be applied. Consequently, the participants during the interviews in the First Deploy got a change in Gerrit with robot comments and then they would fix the suggestions locally.

The reason behind NOT-USEFUL clicks from Black during the First Deploy was due to a bug that reported suggestions on the wrong line. It can also be concluded that if the response to the NOT-USEFUL feedback was made sooner, this bug could’ve been fixed while the deployment was ongoing which in turn would probably have lead to fewer NOT-USEFUL responses.

The NOT-USEFUL clicks, while helping the authors find a bug within the Black MEAN implementation, unfortunately did not uncover any other information regarding the integration of MEAN at Axis. On the other hand, the qualitative data was more informative and helpful in this case to understand more about how developers acted on fixes by robot comments in the First Deploy.

According to the results reported in Chapter 5, D1P1 and D1P2 would like Black to provide fix suggestions in Gerrit while D1P3 preferred to run Black locally. D1P3 reasoned that Black is an uncompromising Python code formatter and as such a developer does not need to think about how the code's style should be implemented. The authors of this thesis agree with the D1P3 that Black does not fit with the usage of MEAN as Black is opinionated, as in it tells the developers exactly what the formatting should be, with very little leeway for configuration of what warning are to be emitted [25]. However Black can still be of interest by fixing the Python code according to the analyzer automatically.

The design of the comment descriptions for the First Deploy phase seems to a large extent to have worked well, though one participant D1P2 made several errors during the scenario, such as inserting fixes in the wrong row. This could perhaps be explained as basic mistakes made due to a more stressful/uncomfortable work environment, as the work is done in a constructed scenario in an interview setting. It is however also possible that this could be due to the descriptions in the comments from the First Deploy being less obvious than first thought.

6.2 RQ2: Supporting Fix Suggestions

To improve the process of acting on fixes suggested by robot comments, support for fix suggestions were implemented for the Second Deploy. This means that the suggested code was moved to a new location, changing the content shown in the comment descriptions. This move lead to three changes to how robot comments were presented, changed descriptions, a SHOW-FIX button and an APPLY-FIX button.

Description: The fix suggestions have been moved so that a button SHOW-FIX instead displays how the code will be changed. The comment description now describe only the reason for the proposed change and also includes a links to more details. These improvements in the Second Deploy lead to a better design of robot comments compared to in the First Deploy. With this design, information is now clearly separated, where descriptions have the actual warning, and SHOW-FIX the solution.

SHOW-FIX: When pressing the SHOW-FIX button, a new window appears where the code is shown without fixes and with fixes. The benefit with SHOW-FIX is that this new way of showing suggestions is much clearer to developers as they can immediately see the change to be made.

APPLY-FIX By pressing APPLY-FIX, the code is automatically changed by the fix suggestion and the user ends up in edit mode, where either the developer publishes or deletes the change. The APPLY-FIX button is quite beneficial as it allows a developer to immediately insert a suggestion into their code, saving time and energy.

6.3 RQ3: How Effective Are Improvements

Generally, support for fix suggestions worked better in the Second Deploy compared to the First Deploy.

After the Second Deploy, it was found that while the fix suggestions and application features were well received, see Figure 5.4. The results in Figure 5.7 together with Figure 5.4 and Figure 5.6 also indicate that while leaning towards positive, the response in general towards MEAN is quite mixed. This was also reflected in the interviews, where once again the overall impression was mixed, but leaning towards more positive as seen in Figure 5.3 and Figure 5.2.

6.3.1 Comment Implementation Issues

When comparing the scenarios in the interview protocols (more details in Appendix A), it is important to note that, as mentioned in A, the protocol changes between the two deployments are quite minor. The biggest changes are the new questions asked regarding the new features, and the inclusion of the new features within the scenario.

One of these new features, the SHOW-FIX button, showed some unexpected results. During the scenarios, three of the four participants did not use the SHOW-FIX function of their own will, instead trying to figure out the intended error using the links provided or asking the interviewer for more information. However, as noted in Figure 5.3 and Figure 5.2, to most participants the suggestions were considered helpful, with one participant D2P2 considering them not helpful as they sometimes gave false positives. So the lack of usage was not due to a lack of appreciation. Rather, during the observations it was noted that the participants simply missed the button.

This might have come from the button just not being visible enough, but could also be due to the participant not wanting to experiment, a notion which might be further strengthened due to participating in an irregular work context, i.e. an interview. The protocol of the interview might also have made the button more difficult to notice as the instructions said to download the code, arguably leading the developer to primarily use their local workstation.

Assuming that the misleading protocol was not the main issue, getting around these misses without giving the developer explicit instructions on what SHOW-FIX does would require calling attention to the button, perhaps by changing the color of the button or the layout of the comment.

The implementation of APPLY-FIX also had issues. While the ability to immediately apply suggestions was considered helpful by participants during the interview, the participants pointed out that working the feature was clunky, as using APPLY-FIX on an issue applies the fix in an edit and puts the developer in edit mode on the change overview page.

Not only is this confusing to developers as no obvious notification that the fix actually was applied is shown, while the only indication that the developer has entered edit mode is a banner near the top of the browser window. Furthermore once in edit mode, there is no easy way to access the inline comment view that's usually used when working with comments. Add to this the fact that this context switch to the change overview happens every time a fix is applied, making it incredibly time-consuming to resolve several fixes.

Resolving these issues would require changing the behaviour of the APPLY-FIX button to be more obvious and straight-forward. The context-switch after every applied fix provides no real benefit and should instead be changed to an inline comment-view where the latest patchset and the current edit are shown in the full-screen diff-view. When pressing APPLY-FIX the application of the fix and the change to edit mode should be made more clear. If the previous context switch issue is fixed that will also help with this problem, though further notifications may be needed.

Additionally, an issue brought up by one participant, D2P4, was that the descriptions for Black specifically did not give any specific information on what the problem was, instead giving a general formatting error message. This could be solved by parsing the output from the Black analyzer, though that would be a lot of work if every type of error were to be checked for.

6.3.2 Comment Implementation Success

While the SHOW-FIX and APPLY-FIX had their share of issues, the new description was mostly well regarded during interviews, especially the new link functionality, seeing much use during the scenario. The one issue mentioned was that Black descriptions specifically were lacking.

6.3.3 Developer Opinions On MEAN

As mentioned in Section 6.3, the response to the MEAN system was mixed leaning positive. To investigate why that is and what can be done to improve the system, a summary of the benefits and drawbacks of MEAN has been produced from the interviews, survey results and not-useful reports:

- Negatives
  - This analysis should be done before publishing the comment for review on Gerrit.
  - A lot of the feedback issued is on less important stylistic errors
  - Conflicting checks
  - Lots of noise
  - Too many false positives
  - Warnings do not match set repository policies
  - The configuration should be branch and repository specific
- Positives
  - Remove otherwise necessary setup time for analysis tools on individual projects
  - Enforces a higher code standard
  - A good visual representation of fix suggestions and applications
  - Might catch errors that were missed during analysis on the local workstation.

Focusing first on negatives, four or even five points can be mitigated by configuring the analyzers better. The "Lots of noise", "conflicting checks", "stylistic error checks" and "warnings do not match set repository policies" issues can all have their impact lessened by making a more stringent blacklist, where "conflicting checks", "stylistic error checks" and "warnings do not match set repository policies" can be removed entirely. Lots of noise can be harder to counteract, as noise might not only be irrelevant categories of warnings, but also warnings on currently irrelevant code. A filter is in place to filter out warnings that refer to code segments not modified in the current Gerrit change [9].

The fourth point alluded to earlier, which is "Too many false positives", can arguably be fixed as checks known to issue false positives can be blacklisted. This might however also remove valid warnings, so this is not a great solution.

The negative point "This analysis should be done before publishing the comment for review on Gerrit" or "don’t publish before analyzing" can be counteracted by allowing MEAN to comment on drafts. This solution does bring the side-effect of a lot more noise, as this would make MEAN send comments for all drafts. Being manually able to tell MEAN when a draft should be checked would be preferable.

Finally, "The configuration should be branch and repository specific". This is a point the authors don’t fully agree with, as while it is true that having more control over the configuration allows for more specialized configurations, it might also lead to the configuration process being more confusing, as each branch of a repository could have a different configuration. Still, it might be worth considering.

As for the positives, it might be a good idea to further improve on the benefits of MEAN. Points such as "Remove otherwise necessary setup time for analysis tools on individual projects" could be further capitalized on by streamlining the configuration process, for example creating a user interface to more easily set up the desired configuration. This also ties in with "A good visual representation of fix suggestions and applications".

As for "Enforcing a higher code standard", it might be interesting to somehow also "encourage a higher code standard", by somehow teaching developers with the comments. The links can be seen as a rudimentary way of doing this.

The last point is harder to improve on, as it is something it already does. What can be done here is really just to improve MEAN in general, especially when it comes to being precise.

Chapter 7

Threats to Validity

7.1 Internal Threats

Experimenter bias As the interviews were performed by the authors of the thesis, who are also the tool builders, an experimenter bias may have been introduced. This means that participants reactions may have been changed, for example made criticism less scathing, with the authors in mind.

Inconsistent testing As the interview protocol was a bit loose in how the interview and especially the scenario should be conducted, only containing general direction rather than step by step instructions, it can be argued that the testing was inconsistent. This in turn allows for the possibility of the participants having differing perceptions of MEAN only based on how the individual interview was conducted.

Attrition This thesis allowed for dropouts during deployment, leading to a distinct attrition bias, which means that there might be an issue with dropouts having an impact on the results. This is however not the only way attrition might affect the results, as the willingness of a developer to interact with the comments is highly individual and not something that is forced onto them.

Selection bias The survey was sent to 56 people, all of whom having interacted with a change involved with MEAN. This sample however may not be representative of the general opinion of the developers at Axis and even less so for developers in general. This selection bias issue is further exacerbated when the fact that 12 of the 56 people asked to participate actually answered,

as this emphasizes the fact that some developers may be more prone to answer surveys, introducing further bias.

7.2 External Threats

Small sample size For multiple parts of this thesis, the sample size has been small with 12 answers to the survey, 7 interviews in total and a smaller deployment size than the previous MEAN thesis. This indicates that new investigations with larger sample sizes might be needed to confirm our findings.

Irregular timeframe As most of the testing was performed during summer, this coincided with a lot of developers being away on vacation, leading to the aforementioned small sample size, as well as a modified work schedule, since there could reasonably be less work planned for this reason. This leads to a work environment that could be argued noticeably differentiates itself from normal operating status. Of course, COVID also adds to this fact.

Chapter 8

Conclusion

This master's thesis project has investigated how a fix suggestion feature might best be implemented in a data-driven program system called MEAN and deployed at Axis. As part of this work, new features have been implemented in the MEAN system such as Black and Clang-tidy analyzers and support for fix suggestions for MEAN in Gerrit.

MEAN was deployed at Axis for 6 weeks, 2 weeks for the First Deploy and 4 weeks for the Second Deploy. In the First Deploy, Black was implemented and added to MEAN, producing comments with fix suggestions in text. While in the Second Deploy Clang-tidy was added and Black was improved based on feedback. Links were inserted for category sources and fixes were removed to instead appear by clicking on a button called SHOW-FIX. Finally, support was added for fixing code via Gerrit, i.e. APPLY-FIX. During these 6 weeks, 35 projects were affected via 257 changes leading to interaction with MEAN from 66 developers (owners, reviewers, cc, listeners and submitters). During the First Deploy MEAN produced a total of 109 robot comments and got 61 NOT-USEFUL clicks in total, while during the Second Deploy a total 1268 robot comments and 98 NOT-USEFUL clicks were produced. The total amount of NOT-USEFUL clicks compared to total robot comments decreased in the Second Deploy compared to the First Deploy. During the First Deploy and the Second Deploy, interviews were carried out to investigate how developer act with fix suggestion, either by robot comments or SHOW-FIX and APPLY-FIX buttons. After the Second Deploy a user survey was conducted and statistics on Gerrit data was computed. The results from the interviews, Gerrit statistics and survey together suggest that fix suggestion support is a good addition to MEAN, as the feedback leant towards more positive. It would be useful to

perform these tests on a wider scale to properly evaluate MEAN, and there is a lot of room for extensions of this projects, leading to lots of interesting directions to take this project in.

8.1 Future Work

The list below is the ideas that can inspire next steps:

- GUI for MEAN (Presentation of MEAN, both Configuration file and Back-end).
- To create a new patchset for the MEAN configuration file when clicking the NOT-USEFUL button. This new patchset will suggest adding the not-useful category to the blacklist in the configuration.
- Reduce number of context switches.
- Apply-all button to apply all fixes, could be quite volatile, could be improved with some confirmation needed.
- Direct access to diff view and apply fix.
- Hyperlinks in checks/category, also for analyzer.
- Manual check on edit.
- Ability to ignore (or to in some way remove) comments.
- Larger test with bigger sample size over longer period.
- Analyzer fixes:
  - Better descriptions for Black, replacements instead of additions (improvements Black).
  - Clang-tidy remove identical checks (especially true when redirection to another check is used, i.e. alias).

References

[1] Lint wikipedia page. https://en.wikipedia.org/wiki/Lint_(software).

[2] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 672–681. IEEE Press, 2013.

[3] Nasif Imtiaz, Akond Rahman, Effat Farhana, and Laurie Williams. Challenges with responding to static analysis tool alerts. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 245–249, May 2019.

[4] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 598–608. IEEE, 2015.

[5] GitHub. Overview of shipshape. https://github.com/google/shipshape.

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

[7] Anton Ljungberg and David Åkerman. Data-driven program analysis deployment. *LU-CS-EX*, 2020.

[8] Anton Ljungberg, David Åkerman, Emma Söderberg, Jon Sten, Gustaf Lundh, and Luke 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.

[9] Mattias Leifsson and Michael Pater. The costs and benefits of acting on program analysis results, 2021. Student Paper.

[10] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 181–190, 2018.

[11] Git-home. Getting started - about version control. `https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control`.

[12] Cloudbees. What is jenkins? `https://www.cloudbees.com/jenkins/what-is-jenkins`.

[13] Avi Silberschatz, Peter B. Galvin, and Greg Gagne. Chapter 18: virtual machines. `https://codex.cs.yale.edu/avi/os-book/OS10/slide-dir/PPTX-dir/ch18.pptx`.

[14] Avi Miller. Containers and how they differ from virtual machines. `https://www.youtube.com/watch?v=98ZNU1KqJfc`.

[15] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[16] Łukasz Langa and contributors to Black. Black home page. `https://black.readthedocs.io/en/stable/index.html`.

[17] Łukasz Langa and contributors to Black. Black author page. `https://black.readthedocs.io/en/stable/authors.html`.

[18] Lukasz Langa and contributors to Black. Black readme.md. `https://github.com/psf/black#used-by`.

[19] Clang: a c language family frontend for llvm. `https://clang.llvm.org/`.

[20] The Clang team. Extra clang tools 13 documentation clang-tidy. `https://clang.llvm.org/extra/clang-tidy/`.

[21] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C. Gall. Context is king: The developer perspective on the usage of static analysis tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 38–49, March 2018.

[22] Lisa Nguyen Quang Do, James Wright, and Karim Ali. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[23] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 332–343, New York, NY, USA, 2016. Association for Computing Machinery.

[24] MEAN developers. Mean open-source repository. `https://gitlab.com/lund-university/mean`.

[25] Lukasz Langa and contributors to Black. The black code style. `https://black.readthedocs.io/en/stable/the_black_code_style/current_style.html`.

Appendices

Appendix A

Interview Protocol

The structure of interview protocol for the First Deploy and Second Deploy is quite similar, it only has been added a few questions (for example in the background) and sub-task in the scenario. Those questions/sub-tasks that are added will use a blue color instead of black.

Background

1. Provide consent paper
2. If audio recording, make participants acknowledge that consent paper has been read and agreed to. If not, ask the participant to send a signature in the form of a picture.
3. What is your name and position/role?
4. How long have you been working at Axis?
5. How long have you been working with software development?
6. Have you in some capacity worked with program analysis previously?
    a. Are analyzers part of your daily workflow?
    b. Any that come to mind as especially useful?
7. Do you have previous experience in using the code analyzer Black/Clang-tidy?
    a. How often?

Scenario

1. Clone repo provided by us, which contains errors
2. Fetch all branches and checkout scenario (git fetch, git checkout scenario)
3. Do a quick check of both code and comments, and give a first opinion of comments
4. Suggest that at least three errors should be fixed, either let participant choose or specify if participant has no clear preference, where at least one is from Black. Ask about general thoughts on comments. Press PLEASE-FIX on all of the comments you intend to fix
5. Fix the code based on comments
    a. At least two of these should be by using show/apply fix (Probably more, since the replacement to be made isn't shown in the comment)
6. Introduce incorrectly formatted code
7. Commit and push to gerrit, on the same change (git commit –amend)
8. Go to gerrit and see your commit.
9. Check if Black/Clang-tidy has a robot comment.
10. Checks that the robot comment refers to the correct region.
11. Add a new change to code for example print('Hello World')/string::str mhm = “”; if(str == "");

During testing of MEAN, we will observe their activities, and ask them to think aloud.

Questions After Scenario

1. How would you rate this workflow compared to how you usually work?
    a. Was it easy or difficult to get into this new workflow?
    b. Was this workflow effective, as in was there little time wasted?
    c. Could you see this new workflow replace your existing workflow? Or add to it?
2. Did you find these comments to be useful?

a. Was there too much/too little context given?
b. Were the comments easy to understand?
c. Any thoughts on how the presentation of comments could be improved?
d. Is Link in comments useful?
e. Do you think that enabling Black to provide fix suggestions would be helpful? Why/why not?

3. What is your opinion on the presentation of suggestions?
   a. Was it easy to understand the change to be made?
   b. Was it easy to understand why the change should be made?
   c. Any thoughts on how the presentation of fix suggestions could be improved?
   d. Was it easy to use the apply fix functionality?
   e. Do you think that enabling Black/Clang-tidy to provide fix suggestions would be helpful? Why/why not?

4. What are your thoughts regarding the interaction with robot comments? What do you think these buttons do?
   a. Was it easy to grasp what the buttons do? (Provide clarification, e.g. “NOT-USEFUL” SHOW-FIX, “apply fix” and PLEASE-FIX, if needed)

MEAN Score

1. What are your thoughts of the following aspects of MEAN: (Answer between 1-5, 1 is bad and 5 perfect, and Why)
   a. Well presented suggestions
   b. Well presented comments
   c. Easy to interact with comments
   d. Fix suggestions are helpful
   e. The fix application is well implemented
   f. I will use MEAN if it has come on the market
2. Are there any issues you had with MEAN you’d like to address, or any improvements you’d like to see implemented?

Appendix B

User Survey

---

* Required

As some of you may know, MEAN has been intermittently running on the Axis Gerrit server this Summer, putting out comments on a variety of issues in several different languages. For those of you who don't know what MEAN is, it's a meta-analyzer system that analyzes files submitted in patches through Gerrit. The MEAN system may run several analyzers on these files depending on the configuration. The results of this analysis will then be sent back to the developer in the form of comments in Gerrit.

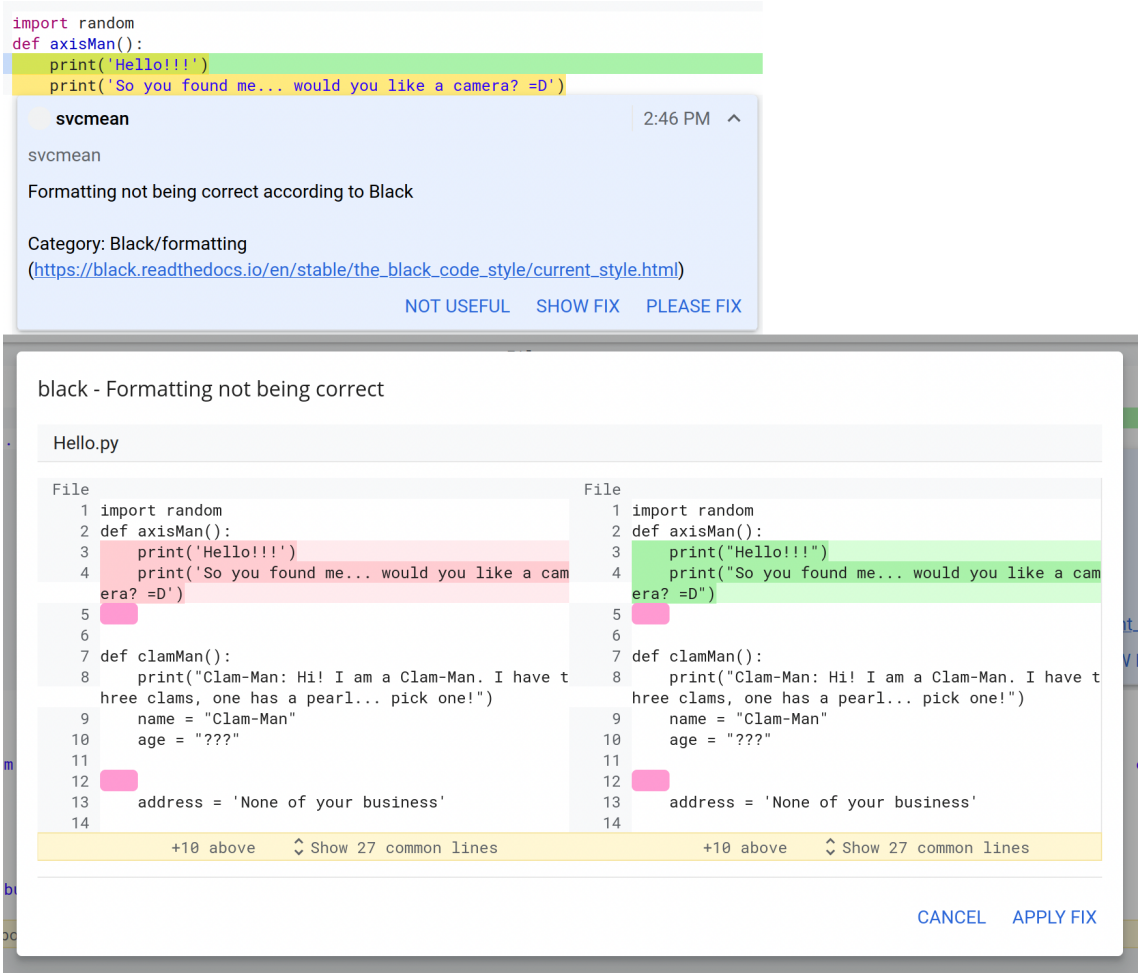
1. Which team are you working in at Axis? *

    ○ Team A

    ○ Team B

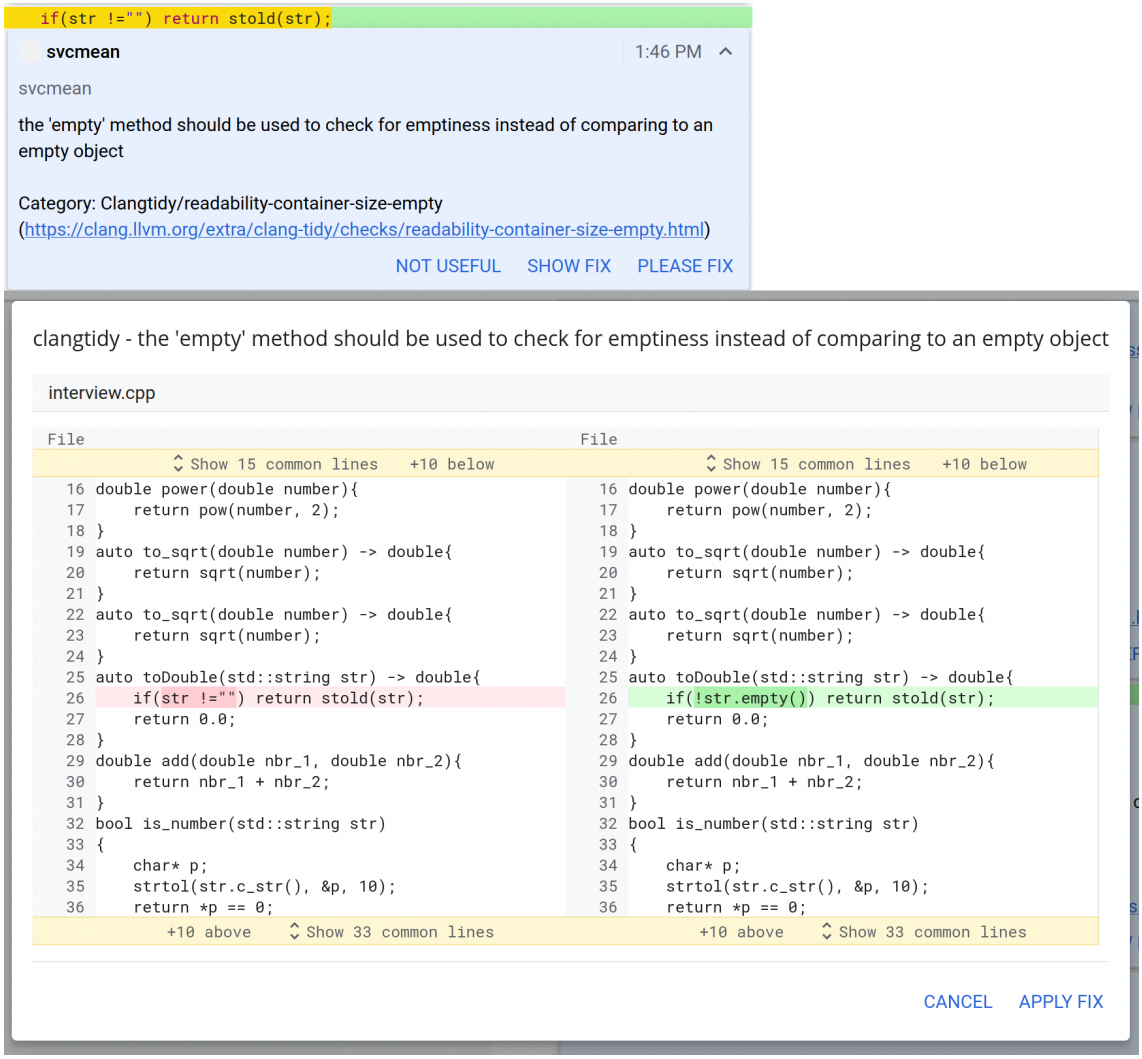
Example of how feedback could look like from the "Black" analyzer

This example is made to provide some insight into how feedback from MEAN might be provided, in case MEAN is new to you or in case it would be helpful with a reminder. The first part (the blue comment and up) shows how a comment may be presented, while the second is what may be shown after pressing the SHOW-FIX button. Both of these are shown to users of the Gerrit code review system. This example is for the python formatter Black.



Example of how feedback could look like from the Clang-tidy analyzer

This is very much like the previous example but for a C++ analyzer called Clang-tidy.



Feedback on MEAN

2. Which analyzers have you received messages from through MEAN? *

   - ☐ Black
   - ☐ Clang-tidy
   - ☐ Hadolint
   - ☐ Ansible-lint

3. What are your thoughts of the following aspects of MEAN: *
   Questions are both for the Robot comment descriptions and the comparison window opened by SHOW-FIX.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
The description is well presented	○	○	○	○	○
Well presented suggestions	○	○	○	○	○
Easy to interact with comments	○	○	○	○	○
Fix suggestions are helpful	○	○	○	○	○
The fix application is well implemented	○	○	○	○	○
Link information is useful	○	○	○	○	○

4. Would you appreciate if MEAN was to run on the Axis repositories? *

   - ○ Yes
   - ○ No
   - ○ Maybe

5. Could you describe why MEAN would appreciated/unappreciated? *

   Your answer

MEAN usage

6. Assume that MEAN is made available in the future, how much would you use the following features of MEAN in your regular workflow? *

	Never	Rarely	Sometimes	Very Often	Always
I would use the SHOW-FIX button to show how fixes will look like	○	○	○	○	○
I would use the NOT-USEFUL button	○	○	○	○	○
I would use the PLEASE-FIX button	○	○	○	○	○
I would use the link information to know why code must be changed	○	○	○	○	○
I would use the Apply Fix button to change the code instead of doing so by hand	○	○	○	○	○

More issues

7. Are there any issues you had with MEAN you'd like to address, or any improvements you'd like to see implemented?

Your answer

45

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2021-09-23

EXAMENSARBETE Data-driven Deployment of Program Analysis Fixes
STUDENT Kevin Andersson, Mohammad Abo Al Anein
HANDLEDARE Emma Söderberg (LTH), Jon Sten and David Åkerman (Axis Communications)
EXAMINATOR Görel Hedin (LTH)

Värdet av lösningsförslag i programanalys

POPULÄRVETENSKAPLIG SAMMANFATTNING **Kevin Andersson, Mohammad Abo Al Anein**

Programanalys kan vara väldigt hjälpsam när man utvecklar kod, då det kan hjälpa utvecklare hitta problem som annars inte skulle upptäckas, och även hjälpa med att undvika vanliga misstag under utveckling.

Men att använda analyserare kommer ofta med problem, såsom att behöva hantera felaktika resultat, förvirrande felmeddelanden och konfigurationer som är unika och måste läras in för varje analyserare som ska användas.

För att motverka dessa problem, kan ett datadrivet programanalys-system användas för att göra det möjligt att kontinuerligt ställa in konfigurationen för systemet under körning. Ett av dessa systemet utvecklat relativt nyligen är MEAN, som står för MEta ANalyzer.

I detta examensarbete, så var målet att undersöka hur en funktionalitet som än så länge ej är implementerad i MEAN, det vill säga lösningsförslag, skulle kunna impelementeras och utvärderas i MEAN. Denna funktionalitet kommer ge MEAN möjligheten att inte bara skicka meddelanden som den gör nu, utan också möjligheten att skicka specifierade förslag på lösningar som automatiskt kan integreras i koden.

Slutsatsen som nåddes i detta arbetet är att förmågan att använda lösningsförslag är uppskattad, men skulle kunna implementeras bättre. Generellt så verkar lösningsförslagen ha förbättrat utvecklarnas interaktion med MEAN-meddelanden, och ger utvecklarna möjligheten att snabbt och enkelt integrera lösningsförslag i sin kod.

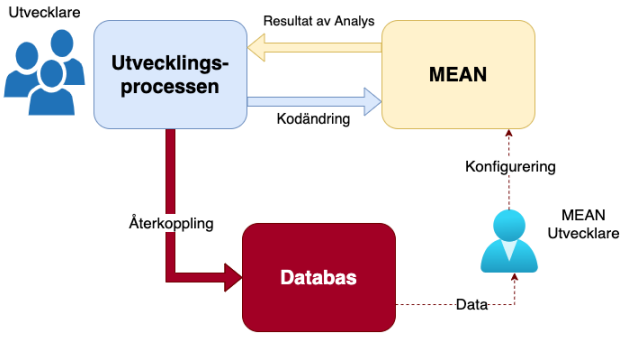


Figure 1: MEAN:s arbetsprocess. När en utvecklare utför en kodändring, så skickas koden automatiskt till MEAN där den analyseras. Därefter så skickar MEAN tillbaka analysresultatet i form av en kommentar. Om utveckarna då anser att kommentaren ej är användbar så kan hen återkoppla till MEAN-utvecklaren som därefter kan konfigurera om MEAN.