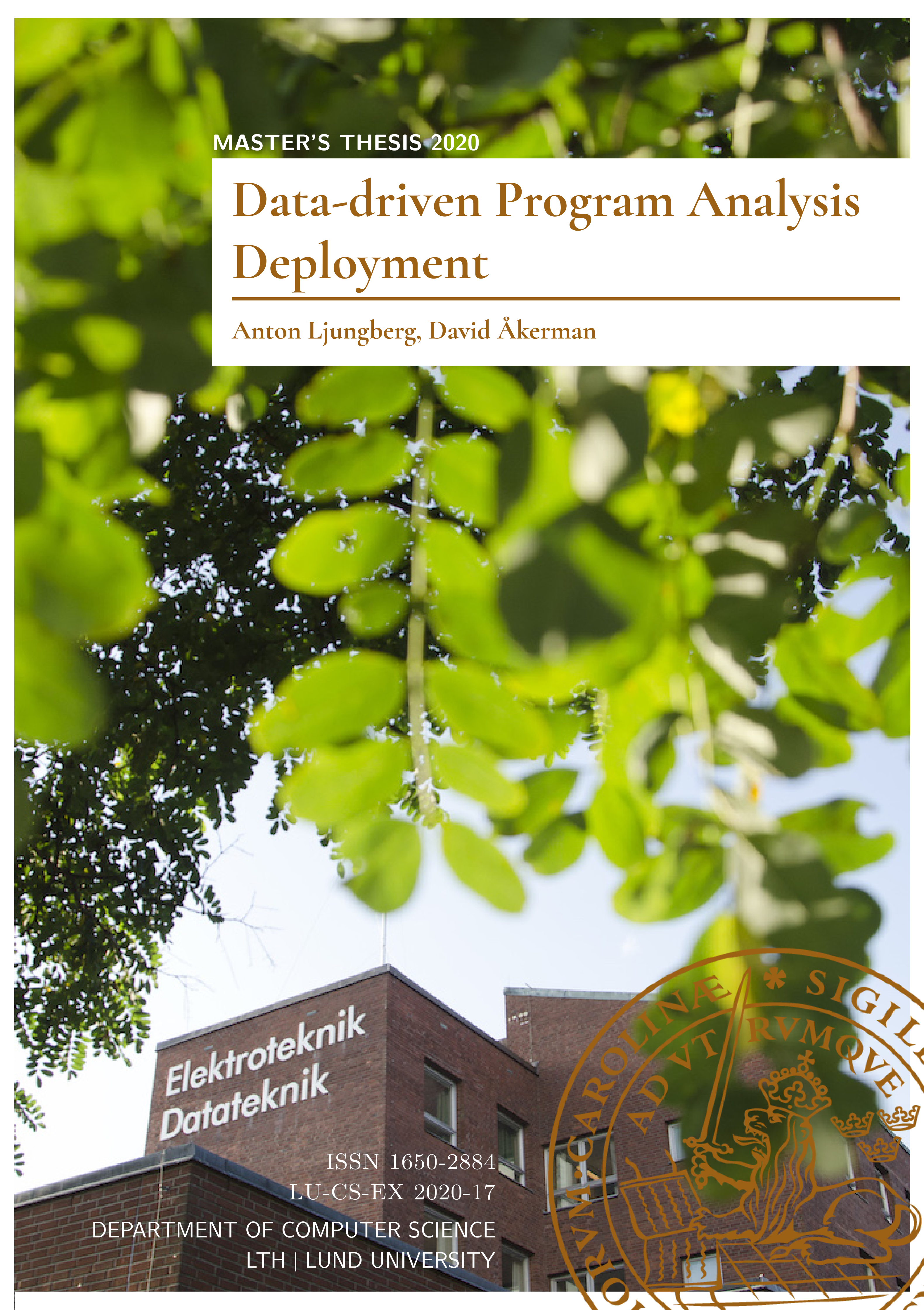

MASTER'S THESIS 2020

Data-driven Program Analysis Deployment

Anton Ljungberg, David Åkerman

ISSN 1650-2884
LU-CS-EX 2020-17

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-17

Data-driven Program Analysis Deployment

Anton Ljungberg, David Åkerman

Data-driven Program Analysis Deployment

Anton Ljungberg
cek11alj@student.lu.se

David Åkerman
dat12dak@student.lu.se

June 22, 2020

Master's thesis work carried out at Axis Communications.

Supervisors: Emma Söderberg, emma.soderberg@cs.lth.se
Gustaf Lundh, gustaf.lundh@axis.com
Jon Sten, jon.sten@axis.com

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

Abstract

Program analysis is useful for reporting code defects that can be hard or time consuming for a developer to find, but usability issues make many developers choose not to analyze their code with such tools. False positives introduce a lack of trust in reported defects. When users can not trust reported defects, they need to spend time on defect validation. Incomprehensible and excessively large results gets in the way of the development process. A promising approach addressing the usability issues of program analysis tools is to adapt the tools to the needs of users by making data-driven improvements.

In this thesis we have created, deployed and evaluated a data-driven program analysis system. We have implemented a system named MEAN (Meta Analysis), together with a handful of protocols for running standardized program analysis within a variety of tool stacks. MEAN has been deployed at Axis Communications with code review as an integration point, where 20 850 program analysis alerts reached the developers. Analyzers were continuously configured in response to user feedback on analyzer results.

Many alerts addressed defects which were not introduced by the change where they were presented. With this stated, users of MEAN fixed one out of ten defects and actively reported alerts as *not useful* once per 40 alerts. When isolating defects introduced by the current change, one out of three defects were fixed and one out of 30 alerts about these defects was reported as *not useful*. Fixing a defect introduced by an older change would often make the current change incoherent, while reporting an alert about such as *not useful* did not have this negative implication.

The evaluation of the deployment verify that noise in analyzer results, including false positives, can be reduced by making data-driven improvements. The evaluation also underline that users of program analysis integrated into daily development shall not be presented with program analysis alerts that are unactionable and redundant.

Keywords: Static Analysis, Data-driven, Program Analysis

Acknowledgements

We want to thank Emma Söderberg, Jon Sten and Gustaf Lundh for their very active supervision of this thesis.

We also want to thank members of the Tools team at Axis for introducing us to existing tools, making it possible to integrate the MEAN system with the Axis tool stack.

At last we want to thank interview subjects, survey participants and users of the MEAN system for contributing to more useful program analysis.

Contribution Statement

System Implementation

Both authors have been active in designing the system. The implementation of initial versions of different system modules has been parallelized. Anton has implemented initial versions of the MEAN main system, storage and publish/subscribe utilities, and MEAN analyzer containers. David has implemented initial versions of MEAN-publisher, Robot-publisher, Analyzer-executor and Gerrit MEAN plugin. Most parts of the system have been reviewed and patched by both authors.

Thesis Writing

Both authors have been contributed to setting the structure of the thesis. Both authors has been active in discussing all topics brought up in the thesis. Anton has written initial versions of chapters and sections: abstract, 1.0, 2, 3, 4.1, 4.6, 5.1, 5.3, 6 (except 6.2.6), 7.1, 7.2, 7.3, 7.4.1, 7.4.2, 7.4.3, 8 and 9. David has written initial versions of chapters and sections: 1.2, 1.4, 4 (except 4.1 and 4.6), 5.2, 6.2.6, 7.4.4, 7.4.5, 7.4.6 and 7.5.

Interviews and Surveys

Both authors have been active in conducting interviews. Anton has written initial versions of the interview protocol and the user survey.

Contents

1 Introduction **11**
- 1.1 Objectives . . . 13
  - 1.1.1 Research Questions . . . 14
  - 1.1.2 Delimitations . . . 14
  - 1.1.3 Risks . . . 15
- 1.2 Overview . . . 15
- 1.3 Glossary . . . 17
- 1.4 Background . . . 17
  - 1.4.1 Version-control Systems . . . 17
  - 1.4.2 Code Review . . . 18
  - 1.4.3 Continuous Integration . . . 19
  - 1.4.4 Containerization . . . 20
  - 1.4.5 Publish/Subscribe Message Handling . . . 21
  - 1.4.6 Program Analysis . . . 21

2 Related Work **25**
- 2.1 Developers on Program Analysis . . . 25
- 2.2 Data-driven Program Analysis . . . 26
- 2.3 Program Analysis Protocols . . . 27

3 Program Analysis at Axis **29**
- 3.1 Axis and Developer Tools . . . 29
- 3.2 Usage of Program Analysis . . . 30
- 3.3 Integration of Program Analysis . . . 31
- 3.4 Organization of Program Analysis . . . 32
- 3.5 Feedback Collection . . . 32

4 Designing a Data-driven Program Analysis System **33**
- 4.1 Requirements and Modularization . . . 33
- 4.2 System Overview . . . 34

4.3 MEAN-publisher . . . 35
4.4 The Main System . . . 36
4.4.1 States Of Analyze Requests . . . 37
4.5 Analyzer Executor . . . 38
4.6 The MEAN Container Protocol . . . 39
4.7 Gerrit Integration . . . 40
4.8 System Communication . . . 42
4.9 Storage Publisher . . . 44
4.10 Configuration . . . 44
4.10.1 Central vs Decentralized Design . . . 44
4.10.2 What Should be Configurable? . . . 45
4.10.3 The MEAN Configuration . . . 45

5 Deploying a Data-driven Program Analysis System 49
5.1 Deployment Stages . . . 49
5.2 Choosing Analyzers . . . 50
5.3 Collecting Data . . . 51
5.4 Code Review Examples . . . 51

6 Evaluation 57
6.1 Evaluation Setup . . . 57
6.1.1 Monitoring Collected Data . . . 58
6.1.2 Metrics . . . 58
6.1.3 User Survey . . . 60
6.2 Evaluation Results . . . 60
6.2.1 Published Robot Comments . . . 60
6.2.2 Responses to Robot Comments . . . 60
6.2.3 Responses to Robot Comments on Changed Lines . . . 61
6.2.4 User Survey . . . 63
6.2.5 Data-driven Changes to Configuration . . . 64
6.2.6 Configurations by Users . . . 65
6.2.7 Feature Requests . . . 66
6.3 RQ3: Not-useful Feedback . . . 66
6.4 RQ4: Why Results Were Not-Useful . . . 67
6.5 RQ5: Fixed Analyzer Results . . . 67

7 Discussion 69
7.1 Did the Data-driven Approach Work? . . . 69
7.2 Other Data-driven Approaches . . . 70
7.3 Design Improvements . . . 71
7.3.1 Additions to the MEAN Container Protocol . . . 71
7.3.2 Configuration of Analyzers . . . 72
7.4 Which is the Best Integration into Code Review? . . . 72
7.4.1 Which Findings to Present to the User . . . 72
7.4.2 Reducing Flooding . . . 74
7.4.3 Suggesting and Applying Fixes . . . 74
7.4.4 Configuration . . . 74

7.4.5 Monitoring User Feedback 75
7.4.6 Monitoring Running Analyzers 75
7.5 What Can Be Shared? 76
7.5.1 Sharing Data 76
7.5.2 Sharing Analyzers 76
7.5.3 Sharing MEAN Components 76

8 Threats to Validity **79**
8.1 Data-analysis 79
8.2 User Surveys 80
8.3 Interviews with Senior Employees 80

9 Conclusions **81**
9.1 Future Work 82

References **83**

Appendix A GStreamer Checkers **87**

Appendix B Interview Protocol **89**

Appendix C User Email Survey **91**

Chapter 1

Introduction

Program Analysis can be a helpful and efficient tool for detecting potential defects in a code base. Automated program analysis can save engineers many hours of labour spent on manually detecting code defects, or even find defects that otherwise would go under the radar and into production code [21]. For projects practicing code review (see Section 1.4.2), program analysis can find defects that code reviewers would have to spend time looking for. As an explanatory example (written in *Python* [11]), it is unnecessary for a human code reviewer to spend expensive time to find the unused import of `math` in Code 1.1, when this can be automated. The found defect can be presented to the developer within the development workflow, e.g. in an IDE, code review or version control, to avoid unnecessary context switching. Ideally an alert with an unambiguous and understandable message, as well as a suggested and possibly automatically applicable fix is presented to the developer such that the unused import of `math` is removed, resulting in code 1.2. After this code reviewers can be involved.

Code 1.1: Python code snippet with unused import

```
1  import math
2
3  def greet(name):
4      print("hello", name)
```

Code 1.2: Python code snippet, after the unused import in fixed

```
1  def greet(name):
2      print("hello", name)
```

Despite these positive effects of using static analysis, software developers in many cases choose not to analyze their code because of usability issues. Static analysis tools report false

positives and floods of alerts, which give developers a noisy and overloading programming experience [21]. Researches looking at questions regarding static analysis tool alerts on *Stack Overflow*, found that the category of question most frequently asked on the subject was how to turn certain alerts off [20]. Seven out of ten developers have their static analysis tools configured only at project kick-off or more seldom [27]. This way of configuring analysis tools has high demands on the knowledge of individuals and leaves no room for exploration. We hypothesize that the cost of turning unwanted alerts off during the whole course of a project is too high, which leads to a missed opportunity of making usability improvements.

In this thesis collective decisions to turn alerts off are continuously taken based on user responses to similar alerts. We accomplish this by designing and implementing a data-driven program analysis system named *MEAN* (Meta Analysis). *Data-driven* means that actions are taken based on statistical data, as a counterpoint to that actions are taken based on personal experience and intuition. The data driving the actions in the MEAN system is data collected from users responding to program analysis alerts. The action driven by this data is configuration of program analyzers. An overview of the dynamics of a data-driven program analysis system is shown in Figure 1.1. The data-driven approach intend to lower the cost of continuously identifying and disabling not useful alerts, during the whole course of a project. This opens a possibility of running analyzers that would discarded if only configured at project kick-off.

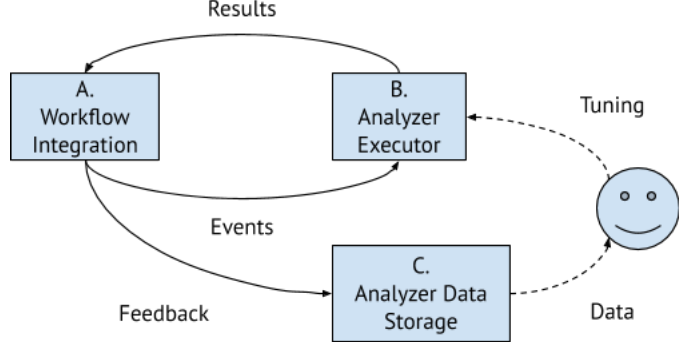

Figure 1.1: An overview of the dynamics of a data-driven program analysis system. Events from A, such as modifications to code, are listened to by B. B starts analyzers and report results back to A. In A the results are presented to the users. Users give feedback to results which is stored in C. Analyzer tuning actions are taken on the data stored in C.

MEAN is integrated into code review and deployed at *Axis Communications* [2] (after this point referred to as *Axis*) to verify that the data-driven approach of running program analysis have the potential to reduce usability issues related to static analysis. The users are presented with code review *robot comments*, which for the unused import of `math` in Code 1.1 would look like in Figure 1.2 and Figure 1.3. During deployment of MEAN we found that false positives were reduced by the data-driven approach. Also categories of defects that were very frequent in some files were reduced because of the overloading of developers, even though they were true positives. Few of the potential defects presented in code review during the deployment pilot study were fixed. The low number of fixes can according to user feedback be explained by the fact that during the deployment pilot study, users were presented with all potential

defects in changed files, instead of only potential defects introduced by changed lines of code. Strategies to keep some true positives by reducing overloading and to present only defects introduced by changed lines of code are introduced in the *Evaluation* chapter.

The data-driven approach has earlier been used to make usability improvements of program analysis tools at *Google*, with the internal program analysis platforms named *Tricorder* [25] and *Tricium* [15]. Apart from the main purpose of making data-driven improvements at Axis, MEAN is designed to be integrated into a diversity of tool stacks. This opens a possibility of creating a cross company community and makes it easier for companies to swap tools while still running the MEAN system.

The contributions of this master thesis are the following:

- This thesis confirms that the data-driven approach of running program analysis can reduce usability issues of program analysis, by using the approach in a new context at Axis.
- This thesis contributes with a design and implementation of a tool-stack-agnostic data-driven program analysis system, including services and protocols.
- This thesis highlight usability factors of program analysis integrated with code review.

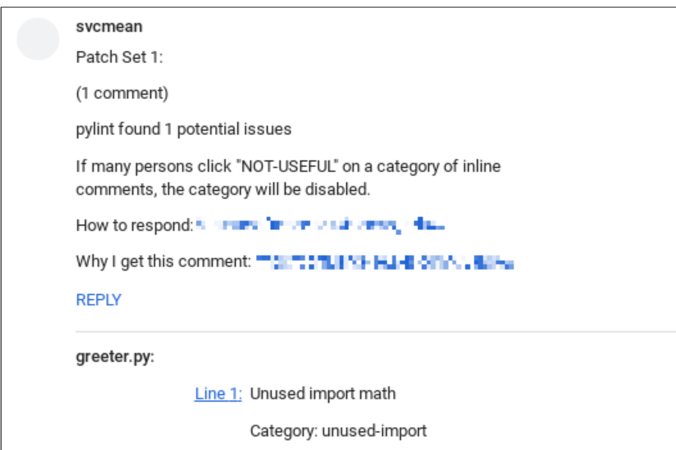

Figure 1.2: A *Gerrit* code review *cover message* presenting a defect in code 1.1.

1.1 Objectives

The main objectives of this thesis are:

O1 To design and implement a basic system for running data-driven program analysis on the Axis developer stack using code review as the workflow integration point.

O2 To run and evaluate a data-driven program analysis deployment pilot study at Axis using the system.

O3 To design a tool-stack-agnostic system which is open source software (OSS).

Figure 1.3: A *Gerrit* code review inline *robot comment* high-lighting a defect in code 1.1.

The main objective (O1) is to design and integrate the data-driven program analysis system with Gerrit and Jenkins, which are used at Axis for code review and continuous integration. The intention with O2 is to deploy the the data-driven program analysis at Axis and let collected data – analyzer findings and user responses to these – drive the configuration of analyzers and be a basis for an evaluation of the system.

By designing a tool-stack-agnostic OSS system (O3), the system will be able to run with a different set of technologies than the ones currently used at Axis. Many companies will have the possibility to use and contribute to the system, which means integrated analyzers and collected data can be shared amongst a larger amount of projects. Better data-driven decisions can be made and initial decision overhead can be reduced.

1.1.1 Research Questions

Summarizing the motive, this thesis aims to answer the following research questions:

RQ1 How can we implement a meta-analyzer system within the constraints set by the established service and workflows at Axis?

RQ2 How can we design a data-driven program analysis system with reusable components?

RQ3 How large is the rate of which analyzer results are found not useful?

RQ4 Why are analyzer results rated as not useful?

RQ5 Were defects reported in analyzer results fixed?

1.1.2 Delimitations

The program analysis system implemented in this thesis will be delimited to fulfill only requirements to be able to run in production at Axis. This means having sufficient functionality

to leave *robot comments* (see Figure 1.4 & Figure 1.5) in Gerrit and collecting data about responses to these comments. The most important objective is to get from *point a*, where there is no running data-driven program analysis system at Axis, to *point b*, where a data-driven program analysis system can be used by a group of developers in production. We name this approach the *a-to-b* approach. Decisions motivated by this approach are described in the chapters about design and deployment (Chapter 4 & 5).

1.1.3 Risks

Prior of starting the project we performed a risk analysis. The risks that we considered were:

Risk 1 The project scope becomes too big.

Risk 2 A too small amount of data is collected to discuss RQ3, RQ4 and RQ5.

(a) A working system is given to a Team too late.
(b) The Team starts using the system too late.
(c) The Team does not use the system enough.

Risk 3 There will be a long time between the system is production ready and necessary infrastructure is acquired.

Risk Management in the Design Phase

To minimize Risk 1 and Risk 2.a the initial focus was to create a minimal viable system. Potential extensions could be implemented at a later time. Meetings with supervisors was scheduled every second week to follow up on progress and discuss future actions.

Keeping the *a-to-b* approach, a secondary objective is to make it possible for as many developers as possible to use and contribute to the system – setting the stage for a future community. This is planned to be implemented by having an infrastructure agnostic design, such that the system can be transferred to another setting at another company.

Risk Management in the Deployment Phase

To minimize Risk 2.a and Risk 2.b we planned to find a team willing to test the system early on and present the system to this team before the deployment pilot study. Choosing a team with an actively developed project that will favour from the chosen static analysis tool is a way to minimize Risk 2.c. Risk 2.c can also be minimized by us as administrators being active in the process and continuously following up on results.

Risk 3 can be minimized by specifying system prerequisites early and make sure orders are made in time.

1.2 Overview

The report is divided into 9 chapters. We begin with an *Introduction* where objectives are stated and concepts necessary to follow the report are described. The next chapter is a summary of *Related Work*. In *Program Analysis at Axis*, we give a picture of how program analysis

Figure 1.4: Two Gerrit robot comments inlined in changed files.

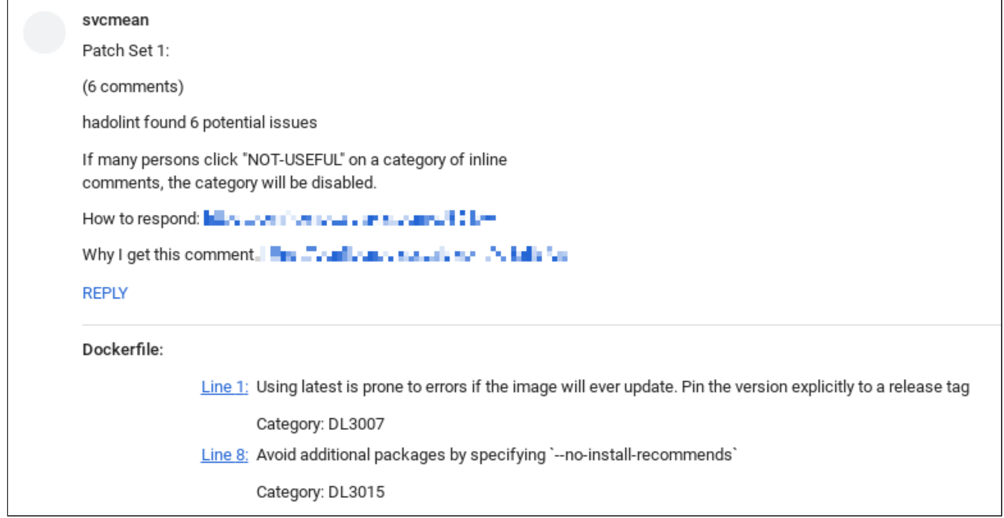

Figure 1.5: An example showing parts of a Gerrit *cover message* posted by the MEAN system. Two out of six robot comments can be seen at the bottom of the image.

have been used at Axis over the last years. In *Designing a Data-driven Program Analysis System* we address RQ1 and RQ2, by presenting the design of the MEAN system. How MEAN was deployed at Axis is described in *Deploying a Data-driven Program Analysis System*. In the *Evaluation* chapter we evaluate the deployment of MEAN and address RQ3, RQ4 and RQ5. The results of the evaluation, the design and the method of running data-driven analysis are discussed in the *Discussion* chapter. *Threats To Validity* are treated in their own chapter. *Conclusion* includes a summary and a description of what work can be done next.

1.3 Glossary

This glossary lists acronyms and words with specific meanings in the context of i.e. program analysis and code review.

Alert A notification regarding a potential defect in a computer program targeting developers

API Application Programming Interface

AST Abstract Syntax Tree

Change Some modification on files in a software repository

CI Continous Integration

Config A configuration file

Finding A found occurrence of a potential defect in a computer program

I/O Input/Output

IDE Integrated Development Environment

MEAN Meta Analyzer

OSS Open Source Software

Patchset A revision of a change. A change can go through an arbitrary number of revisions before it is accepted or abandoned

Program Analysis Static Analysis, Dynamic Analysis, Linters, Code Checkers etc.

1.4 Background

In this section we describe concepts necessary to follow the rest of report. Most descriptions are of types of tools that were integrated into the MEAN system.

1.4.1 Version-control Systems

To be able keep track of modifications of code, you may want to use a Version-control system (VCS). A VCS keep track of different versions of a set of files. A change to a subset of the files are often called a *revision*. Git [9] is one of the most popular VCS systems. In Git a revision is called a *commit*.

1.4.2 Code Review

Code review is an activity where developers review each other's code. It is done to find possible issues, improve code quality and transfer knowledge between developers [26]. Code review can be separated into two different approaches, traditional code review and modern code review. In traditional code review you arrange meetings where several developers review a piece of code. In modern code review you use a tool that is designed for code-review. Gerrit is a popular code reviewing tool that is used by many companies including Axis.

Gerrit Code Review

Gerrit is a web-based code reviewing tool that is tightly built on top of Git. An overview of the workflow in Gerrit can be seen in Figure 1.6. Typical steps of a review:

1. The user upload the change to Gerrit for review
2. Now the change is ready for review and the user waits for comments
3. Other developers review the change and add comments if they find something that can be improved, they submit these comments together with a review label that specify what the reviewer thought about the change overall
4. The user reads the comments and follow the tips or give a reason not to by commenting on the comment, updates the code and upload it as a new patchset. A patchset is a revision of a change.
5. Point 3 and 4 is iterated until the user get +2 as a review label (for more info see next paragraph)
6. The user submit the change and it get merged into the code-base

Reviewers can give the change a score that is called review label. There is five different labels spanning from -2 to 2. What the labels mean is not defined and instead it is the development team themselves that decide the meaning of each label. We list the labels and describe some common meanings:

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."

Even though the meaning of labels is not defined, the labels 2 and -2 has some important functionality. To be able to merge a change you need a 2 on your patch-set and no -2 on the change at all. So label -2 is basically used to block a change from being integrated into the code [8].

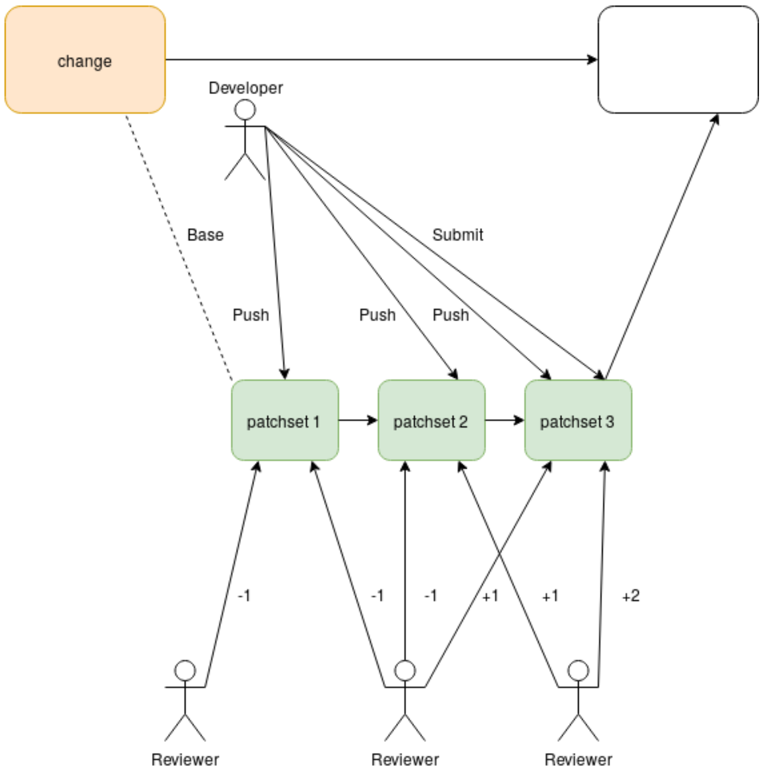

Figure 1.6: An overview of the workflow of a code review in Gerrit

1.4.3 Continuous Integration

Continuous Integration (CI) is a development practice where you try to integrate code continuously, either periodically or when the code has changed. Integrate in this case include things like continuously building the latest code, running unit tests and other steps that can verify that the change did not introduce any bugs. To be able to achieve this effectively you need some software that can automate these steps. Jenkins is one such software that is used by Axis as well as many other companies.

Jenkins

Jenkins is an automation-server. It is used for running scripts which build, test and check applications etc. These scripts are called Jenkins jobs. Jenkins make sure these jobs are triggered periodically or when a specific event occur, e.g. new code is pushed to the code reviewing tool or another Jenkins job has finished. In addition to using ordinary shell script, Jenkins also provide its own scripting framework which is referred to as pipeline script. This pipeline script use groovy and shell as embedded languages. Jenkins also provide a way to store credentials when they are needed in scripts. If any additional feature is needed that is not included in the base installation of Jenkins you have the option to install specific plugins that provide this services, either you create your own plugin or use one that another developer has created.

1.4.4 Containerization

Containerization is an OS-level virtualization method that encapsulate an application in an environment with its own sets of software, libraries and configuration that is isolated from the host. In the context of containerization there are two important parts, an image and a container. A container is a running instance of an image. An image is a binary file that contains the application and libraries and configuration that should be accessible by the application. A container is similar to a virtual machine, the big difference is that containers shares the OS kernel with each other and the host, whereas each virtual machine use its own OS with its own kernel. These OS:es run as guests on the host and it is the goal of a hypervisor program to share the host OS kernel capabilities with these guests. Basically virtual machines has one extra layer of abstraction compared to containers. This abstraction comes with both an advantage and a disadvantage. On one hand you have the option to mix and match different host OS:es with different guests e.g. running Windows guest on Linux host and vice versa. On the other hand having a full-fledged OS for each virtual machine incur an additional overhead that may be mostly unnecessary. Figure 1.7 show an overview of a host system stack with virtual machines. Figure 1.8 show an overview of host system stack with containers. One of the most common containerization solution is *Docker* [5].

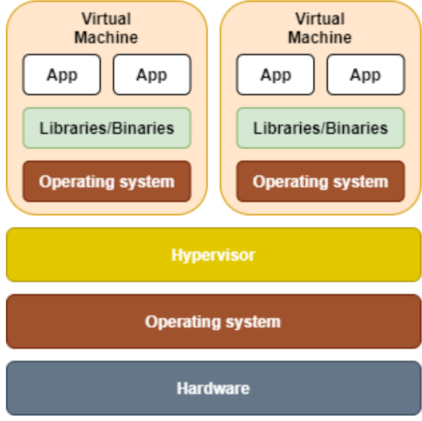

Figure 1.7: A host system stack with virtual machines

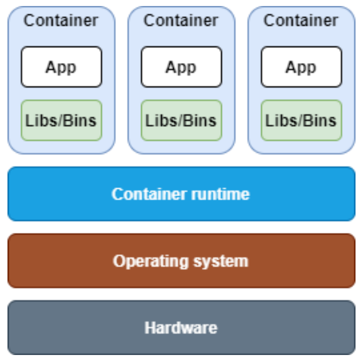

Figure 1.8: A host system stack with containers

1.4.5 Publish/Subscribe Message Handling

Publish-Subscribe is a pattern for handling communication between different components e.g. classes, micro-services, computers etc. The pattern consists of one or several publishers and subscribers as well as a topic. Figure 1.9 show an overview of publisher-subscribe. A publisher send a message to a topic, this message is then accessible to all the subscribers of that topic. RabbitMQ is a system software that can handle this type of communication.

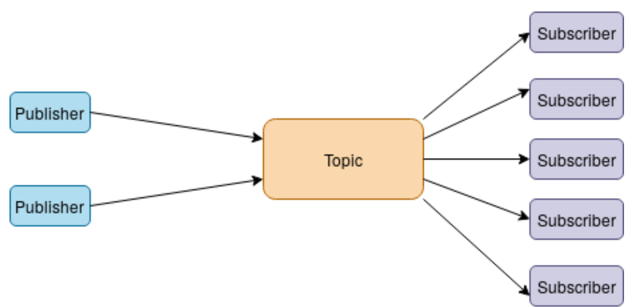

Figure 1.9: Overview of the publisher-subscribe pattern

1.4.6 Program Analysis

Program Analysis is used to automatically analyze programs to find bugs, broken coding conventions, optimization opportunities and other things that impact the behaviour of a program. There is mainly two approaches to do Program analysis, static program analysis and dynamic program analysis. Static program analysis is when you analyze the program without executing the program and Dynamic program analysis is when you do. In our work we have focused on static program analysis and analyzers we have integrated into our system is all static program analyzers. Here is a list of analyzers that we integrated into our system.

Pylint

Pylint is a program analyzer that analyze Python files. It follows the style and conventions recommended by PEP8, but also checks other stuff like unused code. Some of the checks Pylint provide can be opinionated, e.g. how long a line should be or how many attributes a class should have. Pylint is a rather lightweight analyzer, do analysis per file and does not require many dependencies to work. Issues Pylint point out is per line though more lines may need to be changed to fix the issue. Latest versions of Pylint only support Python3.

Flake8

Flake8 is an analyzer that consists of several smaller analyzers, which all focus on style in Python code. Flake8 is rather lightweight, does analysis per file and does not require many dependencies to work. Style issues Flake8 finds point to single lines, even though more lines may need to be changed to fix an issue.

Shellcheck

Shellcheck is a program analyzer that analyze shell scripts. Shellcheck is a rather lightweight analyzer and do analysis per file. Issues Shellcheck point out is per line. Shellcheck support different shells including bash, dash, sh and ksh. Shellcheck give different results depending on which shell it is configured to analyze. By default Shellcheck decide what type of shell it analyze by checking the shebang at the start of the file, if no shebang is found, Shellcheck will point this out. Alternatively you can tell Shellsheck what type of shell it is with a command-line argument.

Coverity

Coverity is a program analyzer that are able to analyze code from several different programming languages e.g. C, C++, Java, Javascript, Python and more. Coverity is designed to find bugs and potential security issues. Coverity is a more heavyweight analyzer and run its analysis on a built program instead of single files. Issues Coverity find may cover several lines of code.

Hadolint

Hadolint is a program analyzer that analyze Docker files. A Dockerfile is used to specify how to build a Docker image. Hadolint is lightweight and only require the files to analyze. Issues Hadolint point out is per line.

Ansiblelint

Ansiblelint is a program analyzer that analyze ansible files. Ansible is an automation tool that simplifies deploying applications. Ansiblelint is lightweight and only require the files to analyze. Issues Ansiblelint point out is per line.

GStreamer coding conventions

GStreamer is an open source multimedia framework and is used by the Streaming department at Axis. The GStreamer project has some coding conventions. These coding conventions include directives on how comments and messages to macro should be formatted.

Comment conventions:

1. Comments should use multi-line comment syntax.
2. Comments that start with "/*" and not "/**" start with a lowercase letter and have no period at the end.

Macro conventions:

1. Strings passed to macro should start with a lowercase letter and have no period. This applies to the macros GST_ERROR, GST_WARNING, GST_INFO, GST_DEBUG, GST_LOG, GST_FIXME, GST_TRACE, GST_ERROR_OBJECT, GST_WARNING_OBJECT, GST_INFO_OBJECT, GST_DEBUG_OBJECT, GST_LOG_OBJECT, GST_FIXME_OBJECT and GST_TRACE_OBJECT.

2. Strings passed to macro should start with an uppercase letter and have a period. This applies to the macros syserror, error, warning, info, g_error, g_message, g_critical, g_warning, g_info and g_debug.

Chapter 2
Related Work

In this chapter we present earlier work related to data-driven program analysis. Articles studied were found using search engines *LUBsearch* and *Google Scholar*. Combinations of keywords searched for were *program analysis* or *static analysis* combined with *data-driven*, *continuous integration*, *integration point* and *false positives*. A first filtering process of all search hits were made by reading abstracts and skimming through content of the articles that were found. This was followed by a snowballing process, meaning that cited articles, judged to contain relevant information, were followed until the relevance of the articles were judged to be too low. This process lead to choosing 16 articles for a closer study. The most relevant related work is presented below.

2.1 Developers on Program Analysis

When we try to address usability issues with program analysis, the opinions of users are essential data. In this section we present related work that highlights opinions of developers regarding usability issues of program analysis.

Imtiaz et al. [20] conducted a study analyzing 280 *Stack Overflow* questions regarding alerts from static analysis tools. The 280 questions were categorized by the authors into one out of 13 predefined challenges interpreting static analyzer alerts. The three categories with the most questions were *Ignore/Filter alerts* (24.9%), followed by *False Positive Validation* (22.9%) and *Problem Resolution Gap* (19.6%). In the *Ignore/Filter alerts* category the author of the question want the analysis tool to ignore certain alerts, in the *False Positive Validation* category the author asks if a certain alert is a false positive and in the *Problem Resolution Gap* category the author of the question do not know how fix a certain alert. The study concludes that developers want more control over the filtering of static analysis results and that makers of static analysis tool should take this fact into consideration by adding highly usable customization for filtering tool alerts on a project level.

Vassallo et al. [27] conducted a study to find out if the developer context where analysis

tools are integrated is having an effect on the configuration of and attention to analysis alerts. The methods used were a survey with 42 participants and a semi-structured interview with 11 participants.

Alerts on found potential defects are often presented at some place in the development workflow, e.g. in editor, code review or continuous integration. On the survey question *"In which step of software development do you usually rely on the suggestions provided by automatic static analysis tools?"*, the alternatives *continuous integration*, *code review* and *local programming* got roughly a third of the answers each. [27] The survey also tells that half of the participants work mostly on projects that are configured only at kick-off, one fifth configure analysis tools monthly, one fifth never do and less than a tenth configure program analysis tools weekly. [27]

Johnson et al. [21] conducted semi-structured interviews with 20 developers to learn more about the attitudes, experiences and opinions of developers regarding static analysis tools. The study concludes that false positives and developer overload are important factors when it comes to developer dissatisfaction with static analysis tools. Except from reducing false positives and developer overload the study suggest a few other things to further improve static analysis tools: enhanced support for team development workflow, intuitive presentation of results, automatic fixes to found defects, and easy configuration.

2.2 Data-driven Program Analysis

Data-driven program analysis has been able to reduce the usability issues with false positives at *Google* and *IBM* [25] [24]. In this thesis we want to verify that the data-driven approach can decrease the number of false positives, by deploying it at Axis. We can also leverage from the prior work done in the area when designing the MEAN system.

Platforms for running data-driven program analysis has been created at *Google*. Sadowski et al. has in an article summarized their experiences building and deploying one of these platforms, named Tricorder [25]. The Tricorder architecture is centered on a driver job starting when notified of a change. This driver start analyzer jobs in three stages: a first stage, when only changed files are known; a second stage, when dependency information is known after a dispatched job has reported build targets affected by the change and a third stage where the affected targets are built in a dispatched job. In the first stage simpler linters are run, in the second stage analyzers related to the build system are run and in the third stage compiler based analyzers that need access to the abstract syntax tree are run.

Sadowski et al. [25] also presents their philosophy on creating a static analysis platform. Ideas like data-driven improvements, code review integration, project scope customization, providing applicable fixes, shardable analysis tools and easy analyzer contribution are encouraged. In Tricorder alerts can be reported as not-useful by sending a partly pre-filled form to the person responsible for the analyzer that generated the finding. Tricorder finds an average of 14 potential defects per *change list* (which is named *patchset* in Gerrit). Reviewers ask the owner of a change to fix an average of 2 out of these and an average of 0.14 findings per change list are reported as not useful. Noteworthy is that the median of defects per change list is 1 and the maximum 5000. A decreasing trend of analyzer warning violations can be seen when looking closer at several types of findings in the code base.

Shipshape [13] is an archived open source program analysis platform project, initiated by

Google. The project `README.md` reads: *"Shipshape is a static program analysis platform that allows custom analyzers to plug in through a common interface"* [13]. Shipshape analyzers are bundled as Docker images that implement a remote procedure call [12] API. The analyzers in Shipshape are run in two stages, *pre-build* and *post-build*, corresponding to the first and the third stage described for Tricorder.

Nanda et al. [24] deployed a static analysis online portal at IBM in 2010, which had the feature of collecting user feedback to static analysis results. The main design goal of *Khasinia* was to provide static analysis as a service, with the creation of a portal being a way to skip client tool integration and eliminate installation overhead. The functionality of the portal, named *Khasiana*, can be summarized as: *"Portal users can upload code to the portal where it is analyzed by the three tools. The results of the three tools are merged and aggregated, and then filtered and sorted based on user-specific analysis configuration and user feedback"* [24]. A found defect shown in the portal can be reported by the users as "Invalid", "WontFix" "Confirmed" and "Not Attended". According to feedback of anecdotal form, the system was appreciated for filtering false positives and the ability to view only defects introduced in a certain build.

User feedback is the data that drives actions in Tricorder, Khasania and MEAN. Not only user feedback can be the data driving program analysis, but also source code itself. Monperrus et al. [23] present a system using a statistical method to detect missing and redundant function calls when using software frameworks. Framework usage patterns that violate the majority of the framework usage get detected by the system. Their evaluation of the system showed that 55% of the detections of missing method calls were true positives.

2.3 Program Analysis Protocols

Kern et al. [22] puts a spotlight on the lack of linkage between static code analysis tools, which makes automation of these tools less traceable and more complex than it could be. A set of tools with a common I/O interface make the static analysis automation tool chain more agnostic, meaning tools can exist without needing to have knowledge of in which context they will be used. Kern et al. therefore introduce a static analysis code exchange format, which make static code analysis results comparable by the common format and traceable by containing meta-data.

In this thesis we define protocols for running data-driven program analysis. Following these protocols, leads to data being stored on a standardized format. The standardized format make it possible to take data-driven decisions for any analyzer in uniform manner. Our definitions of protocols and interfaces are also essential to complete the objective of making data-driven program analysis tool-stack-agnostic.

Chapter 3
Program Analysis at Axis

In this chapter we present the current software development landscape at Axis, with focus on usage and integration of program analysis tools. We explored the landscape by conducting three interviews with senior employees involved with program analysis at Axis. The interviews were covering the topics of integration and organization of program analysis at Axis. They also covered experiences from using different program analysis tools and which feedback that had already been collected regarding program analysis.

To find out how program analysis has been used at Axis we conducted three semi-structured interviews [28] with senior Axis employees. The semi-structured interview approach was chosen because the purpose of the interviews was mainly exploratory, but also detail oriented in some areas. The interview protocol followed during the interviews can be found in Appendix B. Interview subjects were selected for being employees involved with program analysis. Interviews with three persons from different departments were judged to be enough to explore the landscape of program analysis at Axis. Interview subjects were found by *snowballing* from contacts of supervisors and the Tools team. The three interviewed employees have been involved in creating a program analysis culture at Axis in the fields of security, kernel development and video streaming development. All of them had over ten years of experience in their specific field. Two of them had been working at Axis for over 20 years.

Statements in Sections 3.2, 3.3, 3.4 and 3.5 are based on the information collected in the interviews. The three interviewed senior employees at Axis are referred to in the text as *S1*, *S2* and *S3*.

3.1 Axis and Developer Tools

Axis Communications AB (Axis) is a technology company with over 3000 employees, founded in 1984 with its main office in Lund, Sweden. Since Axis in 1996 invented the world's first network camera, Axis has been mostly known for network video products. In the last few years Axis has focused on widening their portfolio of network connected products with e.g.

speakers, door stations and access control units. [1]

The R&D organization at Axis consists of over 1150 employees; many who works with software development. This includes different kinds of software development, e.g. kernel development in C, computer vision in C/C++, video streaming in C/C++ and Rust and developer tools written in Java, Python and Go. The different teams at Axis have a lot of freedom when it comes to which developer tools to use. Most software developers at Axis use Gerrit for code review. During the 8 first weeks of 2020, there were more than 850 unique Gerrit users per week, uploading and reviewing code. Jenkins is widely used for continuous integration, e.g. to build, test and analyze code periodically or triggered on events such as submitting a change. Developers use a big variety of editors and IDE:s.

3.2 Usage of Program Analysis

Static and dynamic analysis are part of the software development process at Axis. S3 tells that the need of static analysis varies from case to case, because other methods to minimize the number of defects are at times more efficient. Requirement management, design and especially tests can at times be so efficient in preventing defects that the investment of sorting and handling static analysis findings is not worth the investment. The biggest cost of using static analysis in the development workflow is the triage of results, meaning the process of deciding if reported defects are true positives and how these shall be prioritized and handled. For software with low test coverage, the investment of running static analysis has a better pay off.

S1 says that dynamic analysis do not have the usability issues with false positives that static analysis do. The dynamic analyzer *Valgrind* [17] is widely used at Axis and is often used while running automated unit tests. Embedded systems sometimes do not have the memory resources to run dynamic analysis which cover a big space of defects. In these cases lighter dynamic analysis supported by modern compilers can be used. Modern compilers often have options for adding run-time instrumentation to generated code. Hardware dependent code can make it complicated to run unit tests. In cases when tests and dynamic analysis are not able to be run, static analysis is often a good investment of time.

Most static analysis tools that are widely used in daily development at Axis are tools *"built for zero false positives"* (S2), which find a smaller variety of defects, e.g. *Cppcheck* [4] and analysis built into modern compilers such as *gcc* [7]. Still static analyzers built for generating few false positives can generate a lot of false positives in contexts with specific behaviours, such as when using certain libraries, frameworks or compiler extensions. An example which has generated false positives is *gcc*-specific functionality for predicting branch of execution. Some static analyzers used at Axis are specifically built for contexts with specific behaviours, e.g. *Sparse* [14] for kernel code which uses compiler extensions.

S1 tells about a static analysis tool earlier used for C development at Axis, which reported many false positives. This lead to the source code being full of comments overriding the report of the false positive. For the unused import *Python* example in Code 1.1, an override can look like in Code 3.1. The situation with overrides all over the code base got unsustainable and the static analysis tool stopped being used for C code at Axis.

Code 3.1: Greeter with unused import

```
import math # analyzername: overide unused-import

def greet(name):
    print("hello", name)
```

A static analysis used at Axis since 18 months back is *Coverity* [3]. All three interview subjects agree to that Coverity finds a lot of different more complex security related defects, but comes with the cost of producing some false positives. S3 means that Coverity is originally built for the purpose of security teams going through the results.

3.3 Integration of Program Analysis

The experience from the interviewed senior engineers at Axis tells that with analysis tools there is a trade-off between being able to find many complex defects and not having any false positives. All three interview subjects are unanimous in that analyzers that find too many defects are not suitable for integration into daily used tools. S2 says that analyzers that does one simple thing well and have very few false positives fits a lot better in the daily developer workflow than analyzers that are able to find many complex problems but have many false positives. Developers do not have the time to triage results that they do not fully trust in their daily work. Also more complex analyzers demand a higher domain specific knowledge.

There is a vision at Axis of putting as many tools as possible into the hands of the developers to use in their daily workflow. When more complex analyzers have been integrated into daily development it has led to developers saving time by ignoring reported defects for some months and later triaging the reported defects. More complex analysis tools has earlier been run and triaged by specialist teams to file bugs on the subset of found defects that were judged to be important to fix. This workflow has been extra efficient for components that are not actively developed.

Many program analysis tools at Axis run as jobs on remote machines using the Jenkins CI toolchain and results from analysis are often presented in Gerrit. An example of this is running Coverity Scan before submitting a change in Gerrit. Most of the program analysis results that in Gerrit are presented as a message, referring to an entire Gerrit patchset, containing a link to where the analyzer results can be further investigated.

Program analysis tools are also run as *git hooks*. In this workflow code is analyzed before it is uploaded to a source code repository and the upload is stopped if potential defects are found. This requires that users have the analysis tools installed locally. Historically some users have skipped analysis at this stage, leading to some code being submitted but not analyzed. Analysis tools with a high rate of false positives are not suitable to run in this stage, because it forces the user to spend time on changing things that are not actual defects.

Some developers at Axis also use IDE:s and editors with integrated program analysis tools, often to follow the rules of analysis that run in later stages, e.g. in git hooks and when uploading code to Gerrit.

3.4 Organization of Program Analysis

Knowledge and guidelines about how to use program analysis is shared between teams at Axis, but it is up to every team to choose which program analysis tool set to use. In recent years the integration of program analysis at Axis have been drifting towards a more centralized model. Introduction of more analyzers, sometimes with complicated installation processes lead to more time having to be spent for setup on user machines. S3 says that developers at this point often want tools to just work in the build chain. S1 says that using a centralized model has the advantage of decreasing the risk of the analysis not being done, and is preferred as long as centralized configuration can be done in a reasonable way. The decentralized model has the advantage of faster and more domain specific decision making, while leaving the risk of the job not getting done.

There have been efforts to put program analysis into IDE:s at Axis in an organized way, but the threshold for developers to skip the installation process has been low and the big variety of editors and IDE:s used by Axis employees has not made the organizational task easier. There is still a vision at Axis of centralizing usage of program analysis in IDE:s because of the very close integration to the development workflow and the quick feedback that is training developers to write better code. The organization of program analysis have had an effect on which integration points are used. Integration into the build chain is, because of running on remote machines, easier to deploy for a centralized model of running program analysis. Integration into IDE:s is, because of running on user machines, easier to deploy for a decentralized model.

S3 have experienced that teams are given more time to use program analysis, when they produce metrics which involve projects leaders in the discussion about program analysis. Invalid metrics can though make decision makers lose confidence in program analysis tools.

3.5 Feedback Collection

There has been some feedback collection for Coverity in Gerrit, where developers could tell if Coverity did a good job when analyzing a change. The results of this are not yet evaluated. Otherwise no organized collection of user feedback regarding results from program analysis has been done. Most feedback has been reported through emails and informal discussions, which easily gets lost in the mass of information.

Chapter 4

Designing a Data-driven Program Analysis System

This chapter cover the design of MEAN and how we arrived at this design. We start by describing the design process and describe the early design of our system. This is followed by several sections that describe different components of MEAN. These sections describe what the function of the component is and explain design decisions that has been made and why they were made.

4.1 Requirements and Modularization

When we designed a data-driven program analysis system there were a lot of decisions to be made. The first big decision was, from a macro perspective, deciding which different parts the system shall be composed of and which communication interfaces these parts shall have. We formed system requirements, on the premises that the system should be given a list of changed source code files of an unknown format as input and analyzer results of a well-defined format as output. Provided was also that the analyzer results should be stored in some kind of data storage as well as being shown in a code review tool in some unknown format. The code reviewers should also, as a part of the system, be able to take action and report individual results as "NOT-USEFUL", and this action should be stored in the data storage. We found the system to have the following essential requirements:

R1 Transform change data to a well-defined format.

R2 Decide which analysis to be done.

R3 Start analyzers given well-defined change data.

R4 Assign computing resources to analyzers.

R5 Persistently store analyzer results.

R6 Publish analyzer results in a code review tool.

R7 Persistently store user feedback to analyzer results.

The responsibilities to fulfil the requirements above were then divided into the following modules:

M1 An *analyzer executor* module responsible for assigning computing resources to analyzers and start them (R3 & R4).

M2 A module (later named *MEAN main system*) responsible for deciding which analysis to be done(R2).

M3 A module (later named *MEAN publisher*) listening to change events acting as a client to the M2 , responsible for transforming change data to a well-defined format (R1).

M4 A *storage publisher* module responsible for publishing analyzer results and user feedback to a data storage (R5 & R7).

M5 A module (later named *robot publisher*) that can publish results to the code review tool (R6).

All of the modules are represented in Figure 4.1, and will be explained in more detail in the following sections. Modules *storage publisher* (M4) and *robot publisher* (M5) act as adapters to the storage and review tools respectively, and are naturally specific for each storage and review tool. Also the *MEAN publisher* (M3) is specific to how code changes are received and represented. The *analyzer executor* (M1) will in its simplest implementation just execute the analysis on the host machine. It can also be used to execute analysis on remote machines and should in this case have the capability of balancing the load of executing many analysers between the machines. This means that the *analyzer executor* is somewhat dependent on infrastructure. The *MEAN main system* (M2) can communicate with all other modules on a well-defined format and does not have to be changed depending on the tool stack.

4.2 System Overview

The MEAN main system could run as an application that was started when you needed it or it could run as server that serve requests continuously. Both options has its pros and cons. Running as an on-demand service has the advantage that it is not dependent on a state, if a failure should occur during the execution of the application, only that instance of the running application is affected. Also having a process for each request is more horizontally scalable because you can put the processes on different machines. Cons are that there is more overhead and it is difficult to monitor the state of the system. For a server solution the cons and pros are reversed. Our system use the server solution because we assumed that our MEAN main-system would be rather lightweight which meant that we did not need it to be horizontally scalable, also we preferred the system to be easier to monitor over the extra reliability.

The system consists of four micro-services and a scalable number of analyzers. The services are:

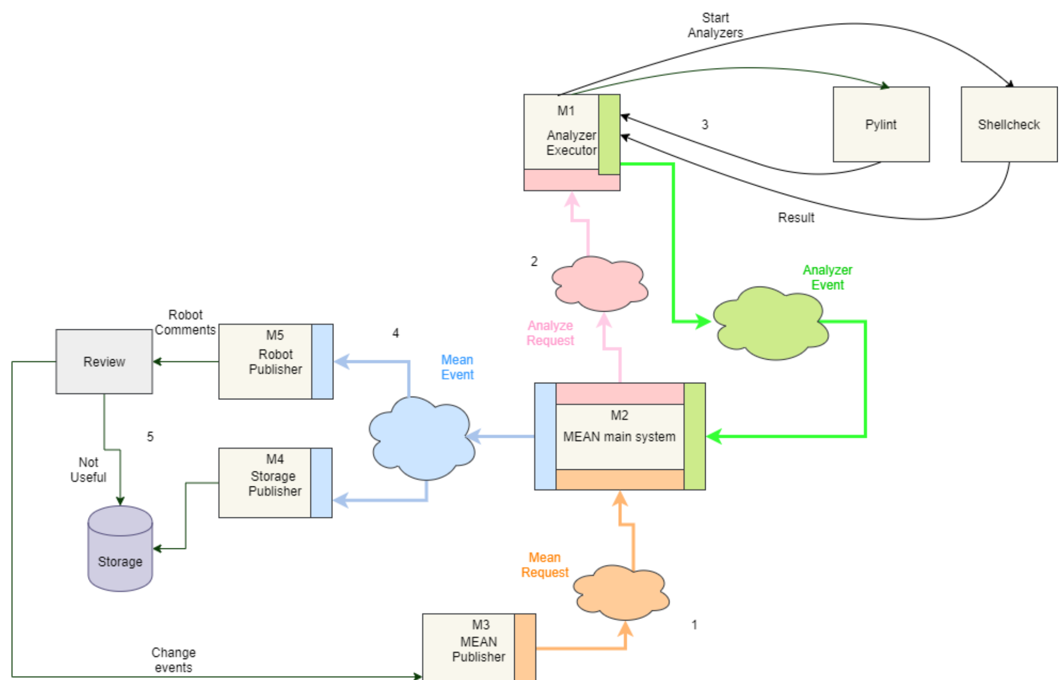

Figure 4.1: An overview of the general design of the MEAN system, the clouds represent message protocols, e.g. RabbitMQ and Kafka. They are different colors to signal that they may not use the same message protocols.

- MEAN-publisher (Section 4.3)
- MEAN-main-system (Section 4.4)
- Analyzer-executor (Section 4.5)
- Robot-publisher (Section 4.7)

The MEAN-publisher is listening for code changes. If a code change occur, data needed for the analysis is transformed to a defined format and sent to the MEAN-main-system. The MEAN-main-system keep track of the different messages and decide which analyzers to run and which files to analyze. Analyzer-executor start the analyzers and provide them with code and other data that is needed for the analysis. Robot-publisher publish the results from the analyzers.

4.3 MEAN-publisher

The MEAN-publisher service is a Python program that executes in a Docker container as a Jenkins job. Gerrit provides a SSH API called stream-events which let you listen for events that occur on Gerrit. This is of interest for us, more specifically info about which changes are pushed to Gerrit. At the start we thought about using this together with the Gerrit rest API to trigger our system as well as getting other info that was needed to find the files to analyze e.g. project name and name of the files that was modified. A bit later we found a Gerrit

trigger plugin for Jenkins. With this plugin you could configure what type of event a Jenkins job should trigger on and Jenkins also put data from the event message into environment variables which you can access in your Jenkins script. This plugin means that we do not need to write code that filter events but on the negative side we become dependent on Jenkins. Jenkins is a big part of the Axis CI pipeline, we therefore thought it was appropriate to use it for our trigger service. Also because of the modular nature of our system it does not matter that much that the trigger is dependent on Jenkins because only this service is dependent on Jenkins. The design decisions to run the Python script in a container was made because our script needed some Python libraries that necessarily was not available on the Jenkins machines.

The MEAN-publisher send MEAN-requests to the main system. Basically a MEAN-request is a request to the main system that a set of files may need to be analyzed. The MEAN-request contain the fields:

1. request-id
2. paths
3. source-context
4. config

The request-id is an id that is generated by the main system that is used throughout the pipeline to keep track of analyzes. The field paths is a list of paths to files that should be analyzed, source-context is implementation-specific data that is required to be able to do an analysis but which the main system does not care about. The source context is added by the publisher and the main system make sure it is appended to the other messages. *config* contains local configuration for different analyzers.

4.4 The Main System

The main and heart of the system is the micro service MEAN-main-system. Every event and request of the system travels through this service. The responsibilities of this service are:

- Merge local configuration and global configuration to a new configuration
- Use the new configuration to determine when to send an analyzer request and what the analyzer request should contain
- Emit MEAN-events that describe what is currently happening in the system and results from the analyzer-executor
- Check that different events for a request happen in the correct order

Analyze-request is a message sent from the main system to the analyzer-executor when an analyzer should analyze some files. The Analyze-request contain the fields:

1. request-id

2. analyzer-name
3. Docker-image
4. paths
5. source-context
6. timeout
7. blacklisted-categories

The request-id is the same id that is included in the MEAN-request. The analyzer-name is the name of the analyzer. The Docker-image is the name of the Docker-image that contain the analyzer and is used to be able to run the Docker image. The paths is a list of paths to files that the analyzer should analyze. The source-context is the same data that is included in the MEAN-request. The analyzer-executor may need this to be able to do the analysis e.g. it can contain info that is needed to retrieve the code. The timeout specify how long an analyzer is allowed to take before a timeout. The blacklisted-categories list categories of checks that the analyzer should not do when analyzing.

MEAN-event is a message that is emitted from the main system every time a MEAN-request or analyzer-event is received or when an analyzer-request is sent. The idea is that by listening for MEAN-events you are able to find out what the main system is doing. Our micro-service that post inline comments to Gerrit use this to be able to get results the analyzers output. A MEAN-event contain the fields:

1. event-type
2. analyzer-event
3. analyzer-request
4. MEAN-request
5. info

Event-type specify what type of message MEAN-event holds. The fields analyzer-event, analyzer-request, MEAN-request are each optional and only one is filled with data at a time and it is dependent on the event-type. Info is a string that can be used to provide extra info about an event occurring. In our implementation info is filled with an error message when a merge of the local- and global configuration fail.

4.4.1 States Of Analyze Requests

To be able to keep track of where in the process of handling an analyze request the system currently is in, we introduce states that the analyze requests can be in. There are six states an analyze request can be in: Scheduled, NotRelevant, Started, Error, Timeout and Result. Files that the main-system decide that no analyzer should analyze lead to a NotRelevant state. For files that should be analyzed an analyze request is sent and the state is set to Scheduled. When

the analyzer is started the analyze request is set to the state Started. When the analyzer has been run the state is set to one of three states. State is set to Error if an Error occur during the analysis, if a timeout occurred, the state is set to Timeout and if neither of these cases occurred the state is set to Result. Figure 4.2 give an overview of the state transitions.

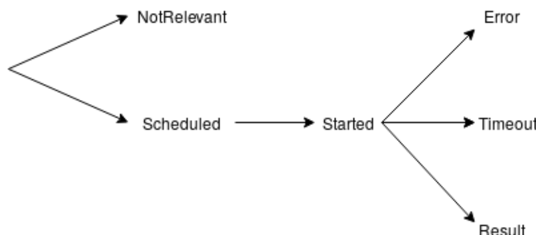

Figure 4.2: State-transition graph of one requested analysis

To make sure that different state changes happen in the correct order. The system save the current state for the different analyze requests. To make the system able to handle an unexpected termination we save the states to disk, so they can be reloaded when the system is started again.

4.5 Analyzer Executor

Our Analyzer executor service is a Jenkins pipeline script that run on Jenkins. This service make use of a Jenkins plugin that trigger a new job when a specific message is found in a RabbitMQ queue. This way we can serve an analyze-request directly. An overview what the Analyzer Executor do when it receives a request:

1. Create the directories `mean/code`, `mean/input`, `mean/output`
2. Write the analyze-request to `mean/input/analyze_request.json`
3. Send a started event to the MEAN main-system to notify that the analyzer is about to start
4. Retrieve the code to be analyzed and put it in the directory `mean/code`
5. Run the analyzer with the `/mean` directory mounted
6. After the analyzer is finished check if the analyze succeeded, timed out or an error occurred and create a response accordingly that is sent to the MEAN main-system.

Analyzer-event is a message sent from the analyzer-executor to the main system when an specific event occur in the analyzer executor. The Analyzer-event contain the fields:

1. request-id
2. analyzer-name
3. event-type

4. analyzer-result
5. source-context
6. info

The request-id is the same id that is included in the MEAN-request. The analyzer-name is the name of the analyzer.

The field analyzer-result contain the result of of the analysis. This type contains a list of notes and a list errors, an error in this case is a string that describe what went wrong. A note is a type that contains the fields:

1. noteid
2. location
3. description
4. category

Noteid is only an id which is used for traceability. Location specify which file and where in the file an inline comment should be posted. Description is a description of the issue and category specify which category the comment is within. Source-context in this message is the same as the source-context in the other messages. Info is a string where you can put extra info if you need, in our case stderr is put there as previously mentioned.

4.6 The MEAN Container Protocol

Inspired by the *Shipshape project* [13] it was decided, at an as early stage as of when formulating the objective of this thesis, that analyzers used by the system were to be run as containerized applications [18]. Containerization of analyzers is accomplished using *Docker* [5].

Different analyzers have different dependencies as well as different input and output formats. The wrapping layer of a software container can handle the dependency issue and the I/O parsing can for each analyzer be built into the container. This way analyzers can be handled by the system in a uniform manner.

The analyzer protocol is very compatible with the message protocol in Figure 4.6. The output of an analyzer container is an *AnalyzerResult* (see Figure 4.6). The input to a MEAN analyzer container consist of a source code file tree and a subset of the fields in *AnalyzerRequest* seen Figure 4.6, more closely a list of files to analyze and a list of categories for the analyzer to ignore (see Figure 4.3).

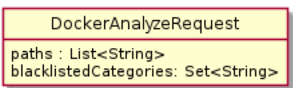

Figure 4.3: An UML representation of the analyze request used as input to an analyzer Docker container.

An analyzer contributor shall implement the following MEAN container interface:

- An analyzer is distributed in a Docker container.
- A volume is expected to be mounted to the container at `/mean`. The file structure of this directory is shown in Figure 4.4.
- In the container a directory with all the required source code is located in `/mean/code/` when container is started – the analyzer container do not have responsibility for fetching source code.
- The input to be able to run analysis is written to `/mean/input/analyze_request.json` and follow the protocol in Figure 4.3.
- The analyzer container is responsible for reporting findings for files listed in the analyzer request while not listing categories blacklisted by the analyze request.
- The analyzer writes its results to the file `/mean/output/result.json`. The result is supposed to follow the `AnalyzerResult`-protocol seen in figure Figure 4.6. Errors that may occur can also be reported in the result file.

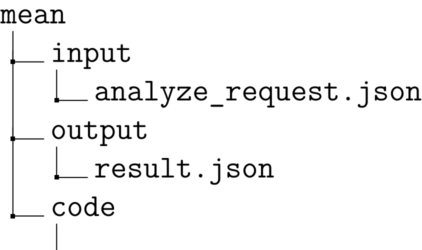

Figure 4.4: File tree of the directory mounted to `/mean` when using the analyzer container protocol.

4.7 Gerrit Integration

The Robot publisher service is its own micro-service. It listens for AnalyzerEvent through a RabbitMQ queue. If there is an AnalyzerEvent and it is of the type result a Gerrit comment is created and posted to Gerrit. Information about what the Gerrit comment should contain are in the AnalyzerEvent object, our specification of range is different to how Gerrit wants it, which means that we do some arithmetic to make it in order.

This part of the system is Gerrit specific but you are able to replace this with your own robot publisher that works in a different way or omit it all together. Most of the design and implementation of this service was straightforward, but there was some important design decisions that impacted the user of the system that was not as easy as initial thought. One of the hurdles was what label you were supposed to post your comment together with (-2, -1, 0, 1, 2). Our initial thoughts was that a robot-comment should use a negative label when some issues has been found and a positive label when no issues has been found. Using -2 in general

seemed like a bad idea because this would require every issue the robot-comment pointed out to be solved before submitting the change. If you could be certain that none of the comments was false positives this would be less of a problem but as stated before this is not the case. We discussed about introducing an option to configure our system to post -2 for certain analyzes but decided that this was outside of the scope of our thesis and instead decided to use -1 for comments with issues. Using 2 for comments with no issues seemed equally inappropriate. Analyzing tools may find many problems in a code-base but there is still problems that only humans are able to spot, therefore it is not good to set 2 after an analysis and 1 was used instead. Later we got the suggestion that we should not use -1 because some may take that as a personal defeat so we changed it to 0 instead. After a bit longer time we got the suggestion not to use 1 either because some developers set a 2 when they see that a change has gotten a 1 from two developers and our service user is not a developer even if it look like it. So in end we used 0 for each robot-comment.

Another question to answer was should we post all issues found for the whole file or should we only post issues that has to do with the lines that has been changed. There is arguments for both ways. Posting all issues is the easiest and most straightforward way, this may not be optimal for the developer and the Gerrit workflow though. When reviewing a change in Gerrit, developers mostly focus on the code that has been changed and give feedback on that. But if we let the robot publish all it's finding for a file it will not work like that. This put the developer pushing the change in a strange place because he will probably get feedback on older code that he may or may not have written at all. This introduce at least two problems, firstly a change should have a commit message that describe what has been added, removed or fixed and a change should by convention be rather small and focus on a specific addition or feature. If the robot give feedback on something outside of the change and the developer want to fix this he have to rewrite the message when he push a new patch-set or he need to create one or more new changes that fix the issues only. Secondly the developer may not have enough knowledge about the other code to decide if a robot-comment is a false positive or not and should in that case not try to fix the issue.

If we only post issues on modified lines the developer may fix the issue in the same change and probably has the knowledge to fix them. Problem is how to achieve this and how this impact the code-base. One idea is to look at changed lines and only post the comment that is within the changed lines. With this solution you will remove many of the irrelevant comments but you will unfortunately also remove relevant comments e.g. if you rewrite a calculation so it does not use a specific variable anymore you may not get a comment pointing out that the variable is unused because the variable is defined outside of the changed lines. A better solution is to compare the issues the analyzer finds to the issues the analyzer finds on the code without the change applied. For issues found outside of the lines modified you can compare them if you add the correct line offset to one of the issue. Find if issues inside the modified lines area are new or not is harder and the easiest way is to assume they are all new. With this solution we will post all issues introduced in the new patchset/change as well as some old issues that hide among the modified lines. But filter out the code may not be what we want at all. Consider that you want your team to use a specific coding convention to make your code clearer and easier to read. If you then have a system that only point out missing convention on new code you may after some time get files that use conflicting conventions. This will lead to code that is less clear and harder to read which is opposite to what you wanted to accomplish.

A Gerrit plugin was developed during this thesis to integrate the MEAN system with Gerrit. This plugin consists of two parts: one part that add the possibility for users to give feedback on analyzer results and one part that let the system get configuration of analyzers for a project.

The feedback part is based on Tricium plugin [16] and include two parts:

1. A button is added to the robot comments in the graphical user interface to report robot comments as "NOT-USEFUL".

2. A rest endpoint that return an URL that "NOT-USEFUL" data should be sent to. This endpoint is only used by the plugin.

The configuration part include two rest endpoints, these are used by the MEAN-publisher to get the local configuration for a project:

1. A rest endpoint that return data that specify if the system is enabled or disabled for a project

2. A rest endpoint that return configuration for the different analyzers for a specific project

4.8 System Communication

The messages sent to and from the MEAN service are shown as an UML diagram in Figure 4.6. For the deployment at Axis they were sent between microservices in *JSON*-format. The MEAN service recieve `MeanRequests` and `AnalyzerEvents`, and send `AnalyzeRequests` and `MeanEvents`.

To make the system less dependent on different communication protocols we implemented the MEAN service as a framework in *golang*. The framework let you implement four interfaces that handle the communication between the services.

1. MeanRequestListener

2. AnalyzerEventListener

3. MeanEventStreamer

4. AnalyzerExecutor

The idea is that MeanRequestListener should be used to listen for MEAN Requests. AnalyzerEventListener is used to listen for events from the analyzers. MeanEventStreamer is used for reporting events in the MEAN system. AnalyzerExecutor is used for running an analyzer in most cases by sending an analyze-request to an Analyzer Executor Service. The implementation developed for Axis' purposes use RabbitMQ for sending requests and listening for events, but Apache Kafka, HTTP or many other communication protocols could be used instead, even combining different communication protocols for different interfaces is also valid. During the thesis, the MEAN framework was used to create a program that ran all analyzers on the host machine.

Figure 4.5: An UML diagram over messages used by the MEAN system.

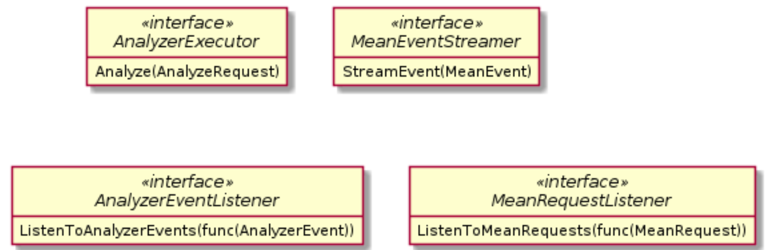

Figure 4.6: An UML diagram over interfaces defined for the MEAN main-system.

4.9 Storage Publisher

An early idea was that the storage publisher would work in a similar fashion to how the robot-publisher would work. It would listen for AnalyzerEvents through a RabbitMQ queue and send this data to a database for storage. Axis already had a solution that logged messages sent on RabbitMQ queues to the storage solution ElasticSearch [6]. So instead of creating a micro-service that handled logging we configured RabbitMQ to log all messages sent through our queues. For the data to be easily readable by a human, JSON are used as the format both for the messages between the services as well as when it is stored.

4.10 Configuration

To find a design for our configuration we wrote down some use cases which we discussed.

4.10.1 Central vs Decentralized Design

1. Should there be one configuration file or several?
    (a) If several, what is relation between the different configs?
    (b) If one, who should handle the config?
2. Where should the configuration/configurations be stored?

Using one configuration file to configure all analyzers for a bigger company like Axis is not viable. A single configuration file with fine-grained configuration of analyzers for many different projects will lead to a very big configuration file that is hard to maintain as well as hard to get a clear overview of. A more coarse grained configuration file will not serve projects that have analysis needs that is very different from the majority.

Finding a good design on where the configuration files should be stored and the relation between these was a harder problem. How fine grained our configuration should be and in what way this configuration should be divided e.g. per project or per user, was problems we discussed. Divide by user give the user much control on how to configure analyzers but our goal was that the system was supposed to be catering to teams or a set of people, so dividing by project made more sense. Configuration per directory and configuration per branch was also discussed. These approaches gives more fine-grain control than per project, but we thought the idea of per project configuration was simpler, more clear and more in line with the a-to-b approach we were going for. To combine configuration per project with configuration per analyzer was also discussed. With this approach the analyzer configuration file included some default configuration that could be overridden by a project configuration. We decided to scrap this idea in favour of a global configuration (Section 4.10.3). A global configuration has the advantage that all the default configuration is in the same file.

In the end we decided for an approach where we have a global configuration and a local configuration that can override configuration specified by the global configuration. The global configuration file is located at the same directory as the main-system. The location of the local configuration file is not specified. Instead it is the responsibility of MEAN-publisher

to retrieve the local configuration and send it to the main-system. This approach is flexible because it does not lock users of the system to a specific way to divide the configuration. The local configuration can be project specific, user specific, etc. For Axis we made the local configuration project-specific (for more info see Section 4.10.3).

4.10.2 What Should be Configurable?

Our approach when trying to figure out what type of configuration we wanted to support, was to start with an empty configuration. Then we added things that we needed and followed it with stuff that we thought could be useful to have in our configuration.

To find the analyzers a configuration option to specify the paths for these is essential. An option to enable or disable an analyzer is also given. Timeout was added to be able to prevent analyzers from taking too much resources and stop analyzers that has frozen. To specify which files the analyzers should analyze the regex option was added.

To be able to disable checks that gives many false positives, we introduce two configurations options: blacklist categories and categories. The option categories can be used in the global configuration and with this you have the option to block, disable, enable and force individual analyzer checks. The option blacklist categories can be used in the local configuration and with this you can disable individual checks.

Many analyzers support other configuration options as well and we tried to figure out how we could support these. The big problem is that these options may be analyzer specific, making it hard to support these in a general way. Let the user put a command-line string e.g. "–shell=shell –check-sourced" or reference a configuration file e.g. ".pylintrc" is a way to let users configure specific analyzers. This approach does not combine well with forcing and blocking. These options has meaning in our system, but may not have any meaning for an analyzer. To make this work we need to somehow merge the configuration file/command-line for the analyzer with the options blacklist categories and categories. Because different analyzers have different formats on their configuration files either the main system need to be made analyzer-dependent to be able to handle the different types of configuration files. Alternatively the configuration files need to be converted to a general format before they are read by the main system. Because we aimed for an a-to-b approach we did not add support for analyzer-specific configuration files/command-lines.

4.10.3 The MEAN Configuration

The final solution of the configuration problem consists of a global configuration and a local configuration. The global configuration is stored in the same directory as the MEAN main system as a yaml file named config.yaml. In this configuration you can configure every analyzer. For an analyzer to be available for usage you need to create an entry in the global configuration for that specific analyzer. An entry consists of a name and the fields status, image, timeout, regex and categories. Each of these fields are mandatory except categories. Status may be one four values:

Block The analyzer is disabled and it cannot be overridden by a local configuration.

Disable The analyzer is disabled, but it can be overridden by a local configuration.

Enable The analyzer is enabled, but it can be overridden by a local configuration.

Force The analyzer is enabled and it cannot be overridden by a local configuration.

Image is the url of the Docker image with the analyzer. Timeout is how long the Docker image is allowed to run before a timeout occur. This timeout is specified in seconds. Regex is a regex that match on files that the analyzer should analyze. With Categories you are able to specify categories of checks that the analyzer should use or not when analyzing files, these flags can be disabled, enabled, forced or blocked. Note that many analyzers have the ability to enable or disable checks on a very fine-grained level which means that a category may only specify one specific check.

Whereas most of the design of the global configuration is defined by us, the local configuration has many design decisions to be decided by the implementer of the publisher. Only the design of which fields should be included is defined. Besides the name, fields that the local configuration contain is timeout, regex, status and blacklist-categories. The field blacklist-categories can be used to specify categories of checks that should not be done. To specify which analyzer configuration to override you need the name, otherwise each of the fields are optional which means that you can override specific behaviour that is specified in the global configuration. How the overriding work is different depending on which field that is overridden. If force or block is set for an analyzer no field is overridden by the local configuration (Table 4.1 show precedence rules). With timeout and status you replace the global setting directly. When overriding the regex both the global and local regex is used to filter out files to analyze. When you blacklist categories in your local configuration you are only able to do that if the category in question is not forced or blocked by the global configuration.

The design decision that the publisher implementer need to decide is how to retrieve or create the local configuration before pushing it to the MEAN main system has both advantages and disadvantages. A big advantage is that you have much flexibility on how this part should be done and can customize it in a way that is most beneficial for your organization, you could have an easy solution where the local configuration for each specific project is found in a file on the same computer as the publisher or you could have a more elaborate approach where you have a tree of local configurations where configurations closer to the root is more global and leafs are the most local configurations. A big disadvantage is that you need to design and implement this yourself which may be trivial but could also be hard and take some time especially considering it may not be obvious what the best approach is for an organization.

Our approach for the local configuration was to use Gerrits inheritance of projects to our advantage. In Gerrit you have the ability create a project that inherit configuration from another project. Configuration of a project can be specified in a file called project.config that reside on the branch refs/meta/config. The project.config of the child and all its ancestors is merged to get the configuration of a project. How the merging works in Gerrit is that a field specified in a child project override the field in the parent. This same mechanism is used by one rest endpoint in our Gerrit plugin but our file is called mean.config instead. By using this rest endpoint you can retrieve the local configuration for a specific project. With this approach users of the system are able to configure the analyzer for their own specific needs by pushing a mean.config to their project.

Table 4.1: $global \wedge local$. Analyzer configuration precedence rules. 1 means enabled and 0 means disabled

local \ global	*block*	*disable*	*enable*	*force*	*not configured*
disable	0	0	0	1	0
enable	0	1	1	1	0
not configured	0	0	1	1	0

Chapter 5

Deploying a Data-driven Program Analysis System

In this chapter we cover how we deployed MEAN at Axis. We deployed MEAN in different stages, a dogfooding pilot, an early tester setup and an exploratory study. There is a section for each stage which explain them. There is also a section that explain how we choose the analyzers to use when deploying (Section 5.2), a section that describe how we collected and monitored data (Section 5.3) and a section that present some code review examples that we experienced during the deployment (Section 5.4).

The deployment of the service can be roughly divided into a few sequential steps:

1. Run a dogfooding pilot at the Tools department with production *Gerrit* and all other parts of the system in a staging environment.
   (a) Run analyzers but not publish any robot comment in Gerrit.
   (b) Notify pilot users and start publishing robot comments.
   (c) Collect user feedback in form of responses to robot comments, e.g. clicking "NOT-USEFUL", and through a chat channel open for free discussion for the pilot users.
2. Move all parts to production environment.
3. An early tester setup by enabling the MEAN system for the Streaming team.
4. Enable MEAN for all software projects at the R&D department.

5.1 Deployment Stages

The first department at Axis to try out the MEAN service was the Tools department, which was where this master thesis were carried out. As the pilot started a chat channel was opened

with the Tools team as members, dedicated for feedback and discussion. The channel was meant to capture user feedback on the MEAN system not captured by responses to published robot comments. The discussions in the chat channel resulted in the detection of a few bugs at an early stage and the settlement of which Gerrit code review labels to use for the robot comments, which is described in Section 4.7. Discussion was done with the Tools team to decide which analyzers to enable. The analyzers that were enabled were Hadolint, Ansible-lint, Flake8, Pylint and Shellcheck (Section 1.4).

To test the MEAN system with a higher load, the system was analyzing all submitted code changes for all of Axis for a period of time, but without publishing any robot comments. This strategy was also used for testing new analyzer containers in a realistic environment.

When the system had been evaluated at the Tools department it was deployed for the Streaming team. A MEAN configuration was uploaded to a parent repository of the projects at Streaming. We asked the Streaming team which analyzers they wanted to be enabled. The analyzers that they chose were Coverity, Flake8, Pylint, Gstreamer-comment-check and Gstreamer-macro-check (Section 1.4).

During a two week period the MEAN system was enabled for all software projects at the R&D department at Axis. We discussed with our supervisors on which analyzers we would enable for R&D. The GStreamer-analyzers was omitted because this was conventions specific for the Streaming team. We decided to only enable one analyzer that analyzed Python code to minimize the risk that developers was overwhelmed with alerts for Python code. The analyzers that were enabled were Shellcheck, Hadolint, Pylint and Ansiblelint (Section 1.4).

Due to *Pylint* already being used by many projects with different configurations respecting the local coding conventions, *Pylint*-categories checking for conventions and style were disabled ahead of the pilot. The other analyzers were run with default configurations.

5.2 Choosing Analyzers

When choosing analyzers to integrate into the MEAN system, we mainly followed 3 different criteria:

A1 Is this analyzer useful for Axis

A2 Is it possible for us to integrate this Analyzer into our system within our time frame

A3 Is the Analyzer a good fit for the system

To find out what analyzer would be relevant for Axis we talked to different people on Axis including our supervisors. When we started Axis was working on integrating Coverity into their Jenkins CI toolchain, which meant we got the tip to integrate Coverity into our system.

Pylint which was the first analyzer we integrated into our system, was used to test if the system worked correctly. This analyzer was chosen because it was easy to integrate and supported essential features like disabling specific checks. That Axis use Python much was a bonus. Two problems that we encountered with Pylint was that it only supported Python3 and our container did not install any extra Python packages. The first problem was partially solved by doing a check that tried to compile a Python file with Python3. Only Python files that compiled without an error was analyzed by Pylint. The second problem was solved

by disabling checks that was about the missing packages. Flake8 was integrated because of similar reasons as Pylint (Section 1.4).

Because Docker and Ansible are commonly used at the Tools department, Hadolint and Ansiblelint were considered good matches. Shell scripts are common at Axis, therefore we thought it was a good idea to integrate a Shell script analyzer. Shellcheck and Checkbashisms was considered, but only Shellcheck was integrated because Checkbashisms missed the ability to disable checks (Section 1.4).

The Streaming team wrote much C and C++ and used analyzers like Clang-format, Clang-analyzer and Coverity. Clang-format automatically format code which means that there is not any result that you can give feedback on. This means that it is not relevant for our system. Both Clang-analyzer and Coverity need extra resources to able to build the program before it is analyzed. This means that these analyzer is harder and more time consuming to integrate than our other chosen analyzers. Also different Axis employees mentioned that the build system on Axis was pretty complex. Because of this we decided Coverity was the only bigger analyzer that we would integrate.

Streaming also wanted macro and comments checked so they could follow a convention set by the GStreamer-project. We created two analyzer that did these analyzes called Gstreamer-comment-check and Gstreamer-macro-check. What they check can be found in Section 1.4.6.

5.3 Collecting Data

The most relevant data or this study is data regarding the published robot comments and the replies and feedback to these.

All messages sent to queues between the micro-services of the MEAN system were saved to a database. Some messages were sent to a *log queue* merely for the purpose of being saved to the database. Information about all robot comments and replies to robot comments were stored by Gerrit and accessible through the Gerrit REST API.

During the deployment pilot study the published comments and "NOT-USEFUL"-clicks were monitored. We did this by creating views of the relevant database content. Published comments and "NOT-USEFUL"-clicks were grouped by category and analyzer, counted and sorted by frequency. At least once a day we looked at the summaries of "NOT-USEFUL" clicks and published comments, to decide if any category needed to be disabled. Analyzer categories were manually configured as disabled if they had a not-useful-rate over 5%. The 5%-rule is closer described in Section 6.1.1.

5.4 Code Review Examples

To set the stage before evaluating the deployment in Chapter 6, we present a few example scenarios experienced during the deployment of MEAN at Axis.

We will start with a scenario where a user fixed several issues presented as robot comments. In Figure 5.1 and Figure 5.2 we see an example of where a developer fixed some issues found by *Pylint*. The user uploaded a new change for review and was presented with 17 issues.

After uploading a new patch for this change, only three issues were left. The fixes lead to 14 fewer robot comments being posted in patchset 2 (Figure 5.2) than in patchset 1 (Figure 5.1).

Now we will give two examples of usability issues seen in this study. In some cases the user got flooded by robot comments, as in Figure 5.3. The 325 inline robot comments disturbed the review workflow, because they could not simply be hidden in the Gerrit UI.

For *Shellcheck* we got many "NOT-USEFUL"-clicks for category SC2039, which finds illegal usages of the `local`-keyword. An example of a published comment with that category is shown in Figure 5.4. Shell scripts are interpreted and different interpreters set different standards. The interpreter cannot be known for sure before run time, so it was guessed using an interpreter directive in the beginning of the shell script file. The actual command-line interpreter used on target, defined the `local`-keyword. The robot comment in Figure 5.4 was a false positive and the category SC2039 was after configured as disabled.

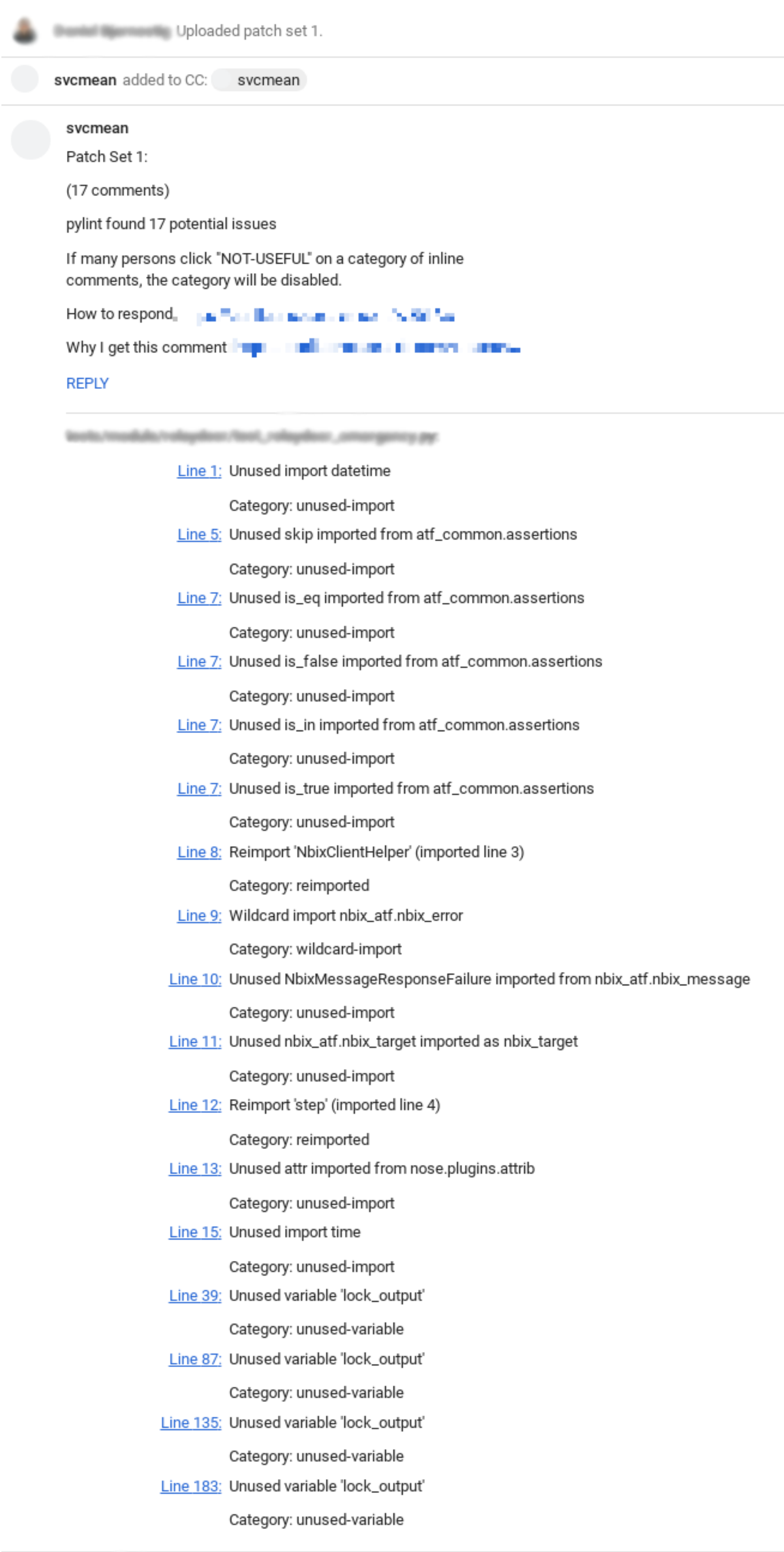

Figure 5.1: A Gerrit code review inline robot comment presenting Pylint issues before they have been fixed.

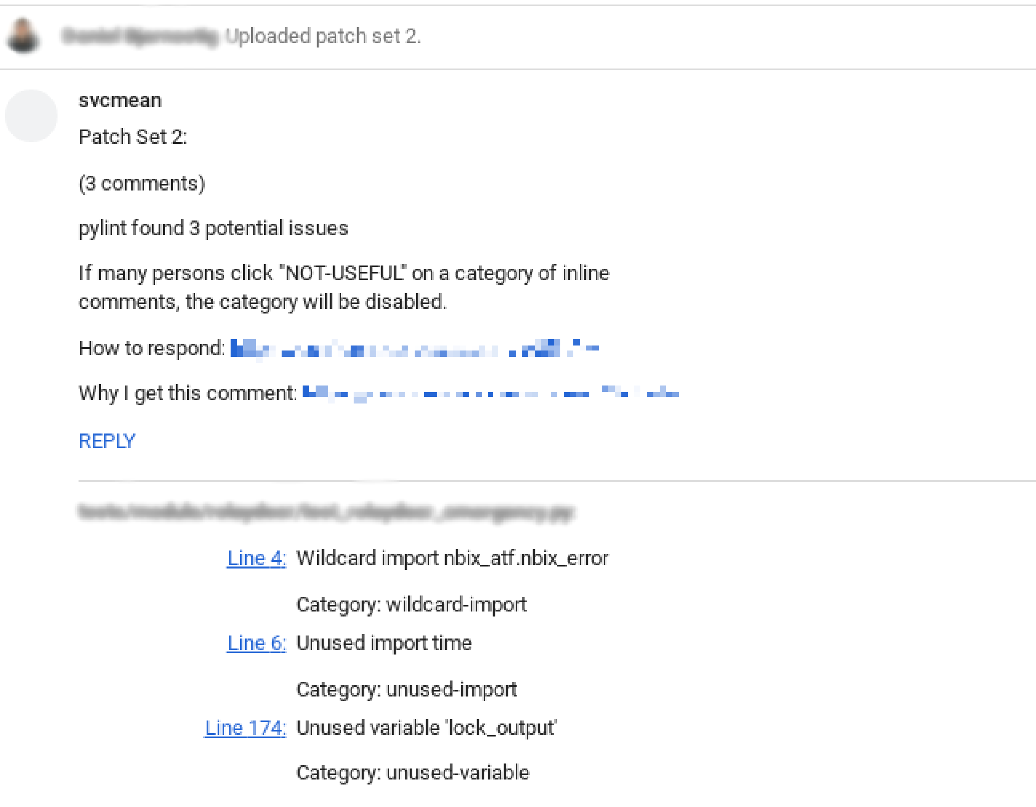

Figure 5.2: A Gerrit code review inline robot comment presenting Pylint issues after they have been fixed.

Figure 5.3: A Gerrit code review inline robot comment presenting a flood of Pylint issues.

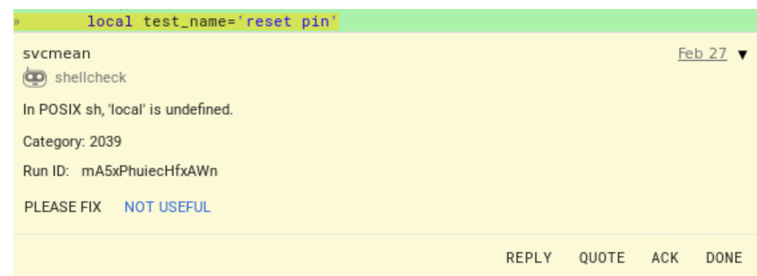

Figure 5.4: A Gerrit inline robot comment that was not-useful.

Chapter 6
Evaluation

In this chapter we evaluate the deployment of MEAN at Axis. We look at which rate users clicked "NOT-USEFUL" (RQ3) and if this rate was high enough to make data-driven improvements. We present some of the MEAN-users explanations to why analyzer results were not useful (RQ4). We also evaluate if the defects presented in code review were fixed (RQ5).

Week	Event
1	Start collecting data for Tools-projects (35 users)
3	Start collecting data for Streaming-projects (50 users)
6	Start collecting data for all projects at R&D(407 users)
8	Stop collecting data for all projects except Tools- & Streaming-projects, run data analysis on all comments & send survey to 20 users
10	Run data analysis on comments on changed lines
11	Stop collecting data for Tools- & Streaming-projects

Table 6.1: A timeline of when MEAN was enabled and disabled for different teams. The number of *users* in team specific projects are the number of unique reviewers and owners of changes in these projects analyzed by MEAN. Data analysis was done at two points on all data collected at these points.

6.1 Evaluation Setup

In this section we describe the method used to evaluate of the deployment of MEAN. The evaluation of MEAN is based on metrics of data collected during the deployment at Axis and a user survey done at a late stage of the deployment. The deployment, survey and data analysis followed the timeline in Table 6.1. MEAN had a total of 446 users, of which 361 used MEAN only during the two weeks it was deployed for all Gerrit projects at the Axis R&D

department. During the eleven weeks MEAN was enabled, we actively disabled analyzer categories which had not-useful-rate over 5% (the 5%-rule is closer described in Section 6.1.1). In week 9 after disabling the system for all projects except Tools- and Streaming-projects we summarized how many robot comments that were collected and which responses users gave to these and sent out the user survey found in Appendix C. The results from the survey lead us to perform data analysis once again, focusing on user responses to robot comments only on changed rows. MEAN was running in production for a short period of time at Axis. Because of this data analysis were done on all robot comments published during this period as a whole.

6.1.1 Monitoring Collected Data

During the deployment we monitored the number of published robot comments and "NOT-USEFUL"-clicks, for every analyzer finding category. As a basic rule, categories were configured as disabled if more than 5% of the published comments of that category were reported as not-useful. This rule is shown in Equation 6.1. We knew that MEAN was going to be deployed at Axis for a short period of time. The seemingly low 5% not-useful-rate is motivated by a focus of making fast usability improvements. The reliability of the not-useful-percentage of a category was judged to be high enough for the pilot study if there were 100 existing robot comments in that category. This also meant that categories with less than 100 published robot comments and at least five "NOT-USEFUL"-clicks were disabled.

$$\frac{\text{NOT-USEFUL}}{\text{ROBOT-COMMENTS}} \geq 5\% \tag{6.1}$$

6.1.2 Metrics

Most metrics used to evaluate the system are simply counts of robot comments and responses. The possible responses to robot comments were "NOT-USEFUL", "PLEASE-FIX", "DONE", "ACK", "REPLY" and "QUOTE". These responses are closer described in Table 6.2. For all responses except "NOT-USEFUL", the user could also report the comment as "resolved" (see Figure 6.1). The "NOT-USEFUL"-button could be clicked by both the owner of the change and the reviewers. A user repeatedly clicking "NOT-USEFUL" on the same robot comment was counted as one click. The responses "REPLY" and "QUOTE" were together counted as *free text* responses. The number of changes and patchsets were also counted to give a picture of how robot comments were distributed.

Estimation of Robot Comment Uniqueness

Unless a reported defect is fixed or configured as disabled, it is found by MEAN again in the next revision. This lead to identical robot comments being posted in patchset after patchset. Duplicates are removed to get the set of *unique* robot comments. *Change unique* comments are unique for a change.

Which comments that were duplicates were not fully traceable after the deployment pilot study, because the line number of a duplicate can change between revisions. We could bound the number of unique robot comments as seen in Table 6.3 and 6.7. A lower bound of the

Button	Why to click button
NOT-USEFUL	The comment is not-useful
PLEASE-FIX	You (reviewer) want the comment to be fixed
DONE	You (owner) have fixed the comment
ACK	You (owner) plan to fix the comment
REPLY & QUOTE	Give feedback on why the comment was not useful

Table 6.2: How users were encouraged to respond to robot comments. When presented with robot comments, users could follow a link that showed this information.

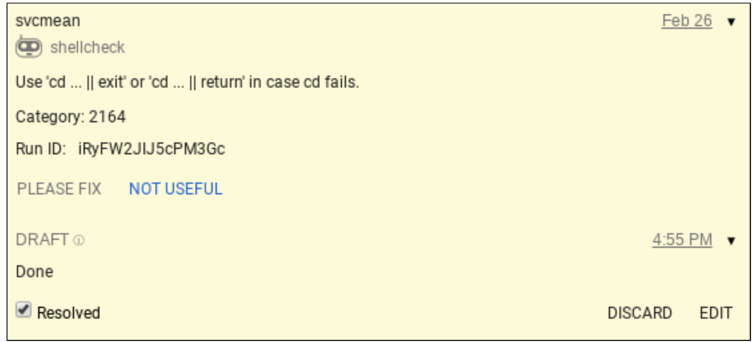

Figure 6.1: Robot comment where the resolved box at the bottom left is checked.

number of robot comments unique to a change was calculated by summing up the number of robot comments in the patchsets with maximum amount of robot comments in each change. A higher bound was found by removing duplicates under the assumption that line numbers did not change. Duplicates were matched on repository name, file name, and line number.

Responses to robot comments can be more sensible to compare to the number *unique* robot comments, because users may not take the time to repeat their responses for every duplicate comment in every patchset. Especially the number of fixed robot comments and "DONE"-replies are sensible to compare to the number of unique robot comments since a robot comment only can be fixed once. This is what we will do in this thesis when stating how many robot comments were fixed.

Estimation of Fixed Robot Comments

The number of fixed robot comments could not simply be counted but was estimated for each change as the number of robot comments in the last patchset minus the number of robot comments in the first patchset. We could have given this metric the verbose name *the decrease of robot comments between the first and the last patchset*. This metric could naturally only be used on changes with at least two patchsets. When summing up this number of fixed comments for all changes we looked only at changes with fewer robot comments in the last patchset than in the first patchset.

A part of the decrease of robot comments between patchsets could be explained by configuring categories of robot comments as disabled. This was adjusted for by removing these comments from the estimate of *fixed robot comments*.

6.1.3 User Survey

We emailed a survey directed to users of the MEAN system to evaluate the deployment. The survey was mainly intended to evaluate why analyzer results were not-useful, but also to collect more general feedback. The survey was sent to 20 users in form of an email (found in Appendix C). Ten users were chosen randomly from owners of changes with fixed robot comments and ten users were chosen randomly from owners of changes without fixed robot comments. The survey was answered by 13 users.

6.2 Evaluation Results

In this section we present results on how users responded to published robot comments and analyzer configurations driven by these responses. We also present the results from the user survey sent out after the deployment pilot.

6.2.1 Published Robot Comments

During the deployment of MEAN at Axis 20 850 potential defects were presented as robot comments in Gerrit (see Table 6.3). The 20 850 robot comments were generated analyzing 485 changes (summarized in Table 6.4, see Section 1.4.2 for definition of patchsets and changes).

After removing duplicated comments from all published comments there were 9283 *fully unique* comments. We found that 10 157 (± 484) robot comments were unique in the change they were published to.

Robot comments	Total
Total Published	20850
Published to changes with one patchset	4527
Change unique	[9673, 10640]
Fully unique, estimate	9283

Table 6.3: Number of robot comments published to Gerrit. In a set of *unique* robot comments there are no duplicates, given that a comment is identified by type of defect and line of code. *Change unique* comments are unique for a change.

6.2.2 Responses to Robot Comments

To find out if issues presented in robot comments were fixed or disabled, we looked at changes with robot comments and more than one patchset. More specifically we looked at the changes individually for each analyzer. This resulted in us looking closer at 265 analyzer-change pairs. The counts of analyzer-change pairs with inline comments and more than one patchset is shown in Table 6.4. If the number of robot comments decreased from the first to the last patchset in these 265 analyzer-change pairs we could assume that the robot comments were

Metric	Total
Changes analyzed	485
Analyzer-change pairs analyzed	765
Analyzer-patchset pairs analyzed	1825
Analyzer-change pairs with one patchset	382
Analyzer-change pairs with robot comments	660
Analyzer-change pairs with robot comments and more than one patchset	265

Table 6.4: An overview of some properties of the Analyzer-change pairs and patchsets that were analyzed using the MEAN system during the first 7 weeks the deployment pilot study. A timeline of the deployment pilot study is shown in Table 6.1.

either fixed or configured as disabled. The trend of number of robot comments per patchset for each analyzer-change pair is displayed in Table 6.6.

After comparing the times when patchsets were pushed, with times when changes to configuration were made, we found that 17 of the analyzer-change pairs with fixed or disabled robot comments was as a result of a configuration change. After a manual inspection of these 17 analyzer-change pairs, it was found that the decrease of robot comments could be partially explained by disabling categories in 9 analyzer-change pairs and fully explained by disabling categories in 4 analyzer-change pairs. In total, a decrease of 175 robot comments can be explained by configuration of disabled categories. If we remove the decrease of robot comments we find that 957 robot comments can be considered as fixed.

Response to robot comment	Total
Not-useful click	572
Please-fix reply	31
Done reply	223
Ack reply	39
Free-text reply	32
Resolved checked	292
Fixed (estimate)	957 (1132 - 175)

Table 6.5: The total count of different responses to getting a robot comment. The actual decrease of robot comments was 1132. After adjusting for the decrease of 175 robot comments explained by configuration we estimate that 957 comments were fixed.

6.2.3 Responses to Robot Comments on Changed Lines

The results from the user survey, indicated that robot comments on unchanged lines were not actionable in the context of a Gerrit change. This motivated data analysis of responses to

Trend	Number Analyzer-change pairs
Decreasing	121
Increasing	23
Unchanged	121

Table 6.6: Trend of the number of robot comments published per patchset for 265 analyzer-change pairs with 2 or more patchsets. An analyzer-change pair has a decreasing trend if the last patchset has fewer robot comments than the first patchset and an increasing trend if the opposite applies.

only robot comments published on changed lines. After 9 weeks of MEAN being deployed there were 3 497 robot comments on changed lines. The count of comments is summarized in Table 6.7. The responses to these comments are summarized in Table 6.8.

Robot comments on changed lines	Total
Total Published	3497
Published to changes with one patchset	575
Change unique	[1661, 1778]

Table 6.7: Number of robot comments published on changed lines. In a set of *unique* robot comments there are no duplicates, given that a comment is identified by type of defect and line of code. *Change unique* comments are unique for a change.

Response to robot comment on changed lines	Total
Not-useful click	129
Please-fix reply	14
Done reply	128
Ack reply	11
Free-text reply	8
Resolved checked	145
Fixed (estimate)	513 (541-28)

Table 6.8: The total count of different responses to getting a robot comment on a changed lines. The actual decrease of robot comments on changed lines was 541. After adjusting for the decrease of 28 robot comments explained by configuration, we estimate that 513 comments on changed lines were fixed.

6.2.4 User Survey

Two out of the participants answering the survey were using Gerrit with the command line interface or an old graphical user interface which did not show the inline robot comments. The survey questions were impossible to answer without having used an interface where robot comments could be seen. Eleven users of MEAN had the possibility to fully answer the survey.

The Context of Program Analysis Alerts

Six out of eleven survey participant mentioned in some survey question that published comments not directly related to a change were not relevant to present in code review. Half of these still agreed with all robot comments they got for their particular change they were asked about. One of these participants wrote that *"comments unrelated to the changed code is something that you want to fix in a separate change"* and as a consequence of this comments on unchanged lines were fully ignored. Similar reasoning was expressed by one other participant.

No matter if the content of the robot comments were agreed with or not, the context where they were presented made them not useful, because they were not actionable in that context.

Why Comments Were Not-useful

The participants of the user survey were asked about what was not-useful in a particular patchset as well as for all robot comments they received. The answers to these two questions were very similar for most participants. In Table 6.9 the reasons for robot comments being not-useful in a particular patchset are summarized. Noteworthy is that only five of the survey participants disagreed with comments in their chosen patchset (see Table 6.10). The answers *false positive* and *against coding standards* were only possible to be given from these five persons, while the rest of the answers could apply to comments agreed with.

Why clicking "NOT-USEFUL"	Mentioned by $N/11$
False positive	4
Comment on unchanged line	3
Low payoff for effort	2
Against coding standards	2
Irrelevant	1
Repeated category of comment in file	1

Table 6.9: Answers to question 6 in the survey found in Appendix C. The question is answered regarding a patchset chosen by each participant.

Participant	Agree	No Opinion	Disagree
P1	2	2	0
P2	2	0	0
P3	4	0	0
P4	2	0	32
P5	14	0	3
P6	8	0	0
P7	9	0	6
P8	1	0	12
P9	114	0	0
P10	1	1	1
P11	19	0	0

Table 6.10: Survey participants were told to look at all robot comments in a single patchset and for each comments tell whether they agreed or disagreed with it. The table shows the number of robot comments each participant agreed or disagreed with.

6.2.5 Data-driven Changes to Configuration

In Table 6.11 a number of robot comments published for each analyzer and the responses to these are displayed. During the pilot of the MEAN system the global configuration was continuously modified according to the 5%-rule described in Section 6.1.1. The analyzers that got enough "NOT-USEFUL"-clicks to make data-driven configurations were: *Pylint*, *Shellcheck*, *Gstreamer-comment-check* and *Gstreamer-macro-check*. In Table 6.13 the disabled categories for each of these analyzers are shown. For both of the *Gstreamer checks* all categories disabled were so due to flooding. For *Pylint* and *Shellcheck* categories were disabled due to being false positives and having low degree of payoff for the effort of fixing them. These reasons for disabling categories are deducted from free-text responses to robot comments, survey answers and manually inspecting changes where robot comments were reported as not useful.

Analyzer	Robot Comments	Not-useful	Resolved	Fixed
Pylint	11333	159	177	614
Gstreamer-comment-check	3185	196	44	48
Flake8	2273	0	0	163
Shellcheck	2204	164	29	44
Gstreamer-macro-check	1117	40	36	59
Hadolint	660	12	6	21
Ansible-lint	62	0	0	8
Coverity	5	0	0	0

Table 6.11: The number of robot comments published for each analyzer and the number of responses to these. Comments not being reposted in the last patchset because of being disabled are are not counted as fixed. These comments are counted in Table 6.12

Analyzer	Whole File	On changed lines
Pylint	135	8
Shellcheck	40	20
All	175	28

Table 6.12: Number of fixed issues that was removed because categories were disabled.

Analyzers with disabled categories			
Pylint	Shellcheck	Gstreamer-macro-check	Gstreamer-comment-check
import-error *wrong-import-order* *line-too-long* *C* R useless-object-inheritance no-member fixme no-value-for-parameter	2086 2154 1117 2002 1090 2004 2039 2162	C0004	C0003 C0004

Table 6.13: The disabled categories for different analyzers at the end of the deployment at Axis. The data-driven approach managed to disable 16 out of these 20 categories of comments that had a not-useful-rate over 5%. Categories disabled for other reasons are *italic*.

Many of the categories was disabled because of the 5% rule, but some were disabled for other reasons (Table 6.13). Pylint has some conventions about which order imports should be placed, based on if they are third-party-packages or not. The *MEAN container* that ran Pylint did not have any extra Python packages installed which meant that Pylint could not know what was a third-party-package and what was not. This made the wrong-import-order giving wrong advice as well as confusing developers. The category import-error was disabled for the same reason. When MEAN was first deployed at Tools, *Pylint* complained a lot about lines being too long. The *Python* developers said that these were not useful especially considering they used another analyzer that already checked that lines was not too long and other types of formatting. Their analyzer accepted a bit longer lines compared to Pylint. We realized that having line-too-long enabled may conflict with other line length conventions and would take away focus from other more useful comments. Therefore the category line-too-long was disabled. The category C in Pylint cover conventions, mostly about formatting. For the same reason as line-too-long we disabled these as well.

6.2.6 Configurations by Users

The ability to configure the analyzers yourself was not used that much by the developers at Axis. Cases where it was used was when they wanted to disable an analyzer or all analyzers completely.

In one case there was project that had Streaming-Projects as parent but did not have to follow the GStreamer conventions. In one project a developer came to us and asked if we could disable Shellcheck for one of his projects because comments covered the code. A team that already used their own configured Pylint wanted to disable all the analyzers for their projects. One person disabled Pylint for a project by themselves. One person disabled all analyzers for a project themselves. Streaming disabled Gstreamer-coment-check themselves because it produced too many issues (Table 6.14).

Project-type	All analyzers	One analyzer
Parent	1	1
Non-parent	1	3

Table 6.14: The number of analyzers disabled by users. Some projects disabled one analyzer while other projects disabled all analyzers. Parent projects configured all child projects.

6.2.7 Feature Requests

When presenting the MEAN system for different teams at Axis we got some feature requests, which are relevant for discussing the best integration of MEAN into code review. The feature request received were:

FR1 Clicking "NOT-USEFUL" shall immediately inhibit getting the duplicate robot comment in the next patch.

FR2 The system shall support analyzer specific configuration files located in projects.

FR3 The system shall be enabled for a specific commit, by putting a "magic word" in the commit message.

FR4 The system shall only report findings for changed lines.

FR5 The system shall support overriding categories of defects at specific positions in files (as in Code 3.1).

6.3 RQ3: Not-useful Feedback

The "NOT-USEFUL"-button could be clicked by both the owner of the change and the reviewers. A user repeatedly clicking "NOT-USEFUL" on the same robot comment is counted as one click. The counts of published robot comments in Table 6.3 and not-useful responses to these in Table 6.5 shows that *the "NOT-USEFUL"-button was clicked once per 40 published robot comments*. For robot comments on changed lines the "NOT-USEFUL" button was clicked once per 30 published robot comments. Users that were active in reporting comments as not-useful, did so at unchanged lines almost at the same rate as on changed lines.

6.4 RQ4: Why Results Were Not-Useful

Most results were not useful for not being actionable in the context of a Gerrit change. This was because these results were related to changed files instead of changed lines of code. These types of usability issue could not be reported by clicking "NOT-USEFUL". Types of usability issues that could be reported by clicking "NOT-USEFUL" were issues that were related to a certain category of defect for an analyzer. An alert being not-useful imply that the user do not want to be alerted about the same category of defect the next time it is found, it does not imply the alert is a false positive.

Categories of analyzer results that addressed conventions and were occurring several times in same files disturbed the code review flow, by forcing the users to see a large number of results at all times when reviewing. Human review comments and other robot comments got lost in the mass. Also the code became harder to read in code review with inline comments on every line. Some categories of analyzer results were not-useful because they were false positives. Either because of going against the present coding standards or presenting defects by making incorrect assumptions of the context. Other categories of analyzer results were not-useful for being too picky. These results had a very low payoff for the work needed to be put down to fix them. The reasons of alerts being not-useful were in summary:

- The alert is not severe enough.
- The alert is disturbing the workflow, e.g. by flooding.
- The alert is a false positive.
- The alert is not actionable.

When talking to users we found that some were more restrictive in clicking "NOT-USEFUL", on the reasoning that the robot comment could be useful in another context. Other users clicked "NOT-USEFUL" as soon as a robot comment was not useful in the context it was presented. If users were not that restrictive in clicking "NOT-USEFUL" more categories might been disabled.

6.5 RQ5: Fixed Analyzer Results

The count of unique defects presented in robot comments is limited by a higher bound of 9283 (see Table 6.3). An estimate of 957 comments were fixed (see Table 6.5). This means that *one out of ten defects presented were fixed*. The count of fixes includes instances of removed code, not only changed code. A user may for example remove a whole file from one patchset to the next if many robot comments are posted in that file. You could argue for that some changes were not closed yet. There were 4527 robot comments in changes with only one patchset. These are per definition change unique comments. If not looking at robot comments in changes with only one patchset, one out of six robot comments were fixed.

For robot comments posted on changed lines one out of three were fixed. If ignoring robot comments on changed lines in changes with only one patchset, about half of the robot comments were fixed. We can also see that users were replying "DONE" and marked robot comments as "resolved" more often for robot comments on changed lines.

Chapter 7
Discussion

In this chapter we discuss the data-driven approach, the integration into code review and the potential of sharing data and system components.

7.1 Did the Data-driven Approach Work?

The two most frequent questions regarding program analysis on *Stack Overflow* are on the subjects: *how to filter alerts* and *validation of false positives* [20]. The data-driven approach used in this thesis address usability issues related to both of these subjects. The collective decisions taken when many users click "NOT-USEFUL" address the validation of false positives. The standardized data-driven configurations done by admins (which in the future can be automated) address the filtering of alerts.

The data-driven approach helped us disable 16 categories of comments that users thought were not-useful. The rule of disabling categories with a not-useful-rate over 5% may seem low, but when we inspected robot comments of categories that were disabled, most comments were too picky, false positives or flooding the users. This means that the usability would be lowered if the not-useful-threshold increased from 5% used during the pilot of MEAN at Axis. We conclude that *the data-driven approach did the job it was intended to do*, which was disabling unwanted categories of alerts.

The collected not-useful-data could only be used to find usability issues deriving from a category of alert. The existence of duplicate comments and not actionable comments could possibly have lowered the response-rate, by overflowing users. Users being new to the MEAN system could also have lowered the response-rate. The not-useful-threshold could possibly be set higher if users were more accustomed to responding to robot comments and if fewer redundant robot comments were posted.

We cannot say that we ended up with a configuration that satisfied every user. The MEAN system generated 20 000 alerts during a seven week period at Axis, while Tricorder used at Google generated 93 000 alerts a day [25]. The shorter time and smaller scale of the deploy-

ment in this thesis lead to some categories of comments being published just a few times, which was not enough to take data-driven actions. The MEAN system would have needed more data to converge to a configuration which produces as few unwanted alerts as possible.

When using Tricorder at Google, both the rule for taking actions on "NOT-USEFUL"-clicks and the actions differ from the pilot of MEAN at Axis. Equation 7.1 shows how the not-useful-rate was calculated at Google (clicking "APPLY-FIX" applied a suggested fix directly in code review). The not-useful-rates were not calculated for categories, but for whole analyzers. At Google, a 10% not-useful-rate lead to the analysis writer being encouraged to change the analyzer. A 25% not-useful-rate lead to the analyzer being turned off. [25]

$$\frac{\text{NOT-USEFUL}}{\text{NOT-USEFUL + PLEASE-FIX + APPLY-FIX}} \geq 10\% \tag{7.1}$$

The approach to calculate the not-useful-rate of MEAN is described in Section 6.1.1. Because of the uncertainty of how often users were going to respond to robot comments, we chose to divide the number of "NOT-USEFUL"-clicks by the number of published robot comments (as in Equation 6.1), instead of the number of responses that implied robot comments were fixed (as in Equation 7.1).

Responses other than "NOT-USEFUL" were a lot less frequent at Axis than at Google. It is possible that the reviewers at Axis did not have the time to get fully accustomed to responding, e.g. "PLEASE-FIX" and "DONE". To find explanations what these response meant, Axis users had to switch context by following a link in the Gerrit *cover message* (see "How to respond" in e.g. Figure 5.2), while an explanation of "NOT-USEFUL" could be seen directly in the Gerrit *cover message*. An approach of calculating the not-useful-rate during the deployment at Axis that would have been more similar to the Tricorder-approach is shown in Equation 7.2. The total not-useful-rate for all comments on changed lines at Axis would with this approach have been $67\% = \frac{572}{572+223+31}$. Because the low frequency of "PLEASE-FIX" and "DONE" this way of measuring not-useful-rate was not used.

$$\frac{\text{NOT-USEFUL}}{\text{NOT-USEFUL + PLEASE-FIX + DONE}} > 10\% \tag{7.2}$$

During the deployment of MEAN we were acting on data grouped by category of defect instead of data grouped by analyzer. This was done to be able to take more fine grained and directed actions. If we would have acted on data grouped by analyzer, *Shellcheck* and *Gstreamer-comment-check* would have been turned off. Instead categories of issues these analyzers produced were turned off. The precision of disabling by category instead of analyzer have the potential to make an analyzer more useful, instead of turning the whole analyzer off.

7.2 Other Data-driven Approaches

There are many other data-driven approaches of running program analysis than the one used in this thesis. The inverse of the approach used in this thesis is possible. Instead of disabling not-useful alerts we could enable useful alerts. Using this opt-in strategy, by starting from no alerts and then enable alerts, will put less initial load on the developers when deciding if results are to be fixed or reported as not-useful. With the opt-in strategy, e.g. human review

comments can be used to find what is often asked to be fixed. Alerts about patterns that are often asked to be fixed are then enabled.

Another possibility of running data-driven program analysis is to use the code itself as data, e.g. to look for abnormal patterns which will generate an alert. An example of this is method calls breaking the rule of majority [23].

7.3 Design Improvements

In this section we discuss what we considered adding to the design of MEAN, but did not implement due to the time frame of the master thesis.

7.3.1 Additions to the MEAN Container Protocol

Due to the *a-to-b* approach the MEAN container protocol was kept minimal. The `DockerAnalyzeRequest` (Figure 4.3) was defined with as few parameters as possible, while still being able to make data-driven improvements by disabling categories of findings. Adding configurable options, will lead to extensions of the MEAN container protocol.

The *Tricorder* [25] and *Shipshape* [13] data-driven program analysis platforms developed at Google have the possibility to find out in which stage an analyzer can execute, described in Section 2.2. There are three stages and they each define the information required for running an analyzer at that stage. The three stages are:

1. Files are known
2. Dependencies and affected build targets are known
3. Targets are built and the AST is known

Analyzers need to inform the environment of which stage they belong to, e.g. Shipshape analyzers do so by implementing a RPC interface with a function dedicated to getting the execution stage [13].

Analyzers in step 3 are compiler based. Shipshape uses a software, named *Kythe* [10], which helps capturing the information about a compiler invocation on a generic format. If having a compiler and a build systems that support emitting the data required to reproduce a build on a generic format, compiler based analyzers that support consuming this emitted data can run.

Amongst the eight analyzers used to different extensions during the deployment pilot study at Axis, seven of them only needed the source code files files to execute. The one exception was *Coverity Scan*, which captures a build invocation to be able to reproduce its own build used for static analysis. With limited time to learn the build system used at Axis we decided to use a script developed at Axis that executed a build according to a *build instruction file* and analyzed the build with Coverity Scan, and with this decision leave the idea taken from the *Tricorder* project [25] of implementing a generic way to run compiler based static analysis within the MEAN program analysis system.

An addition to the protocol where only one build need to be run and a generic way to run a compiler based static analysis is included, should be highly possible to implement within

the MEAN system. Because of the *a-to-b* approach (described in Section 1.1.2), this was not implemented during the course of the master thesis.

Potential ways for analyzer containers in the MEAN system to inform their execution context about their prerequisites for running an analysis are:

- Using *Docker labels* on the analyzer Docker images
- Adding a subcommand or flag to the entry point of the Docker image that returns the information

7.3.2 Configuration of Analyzers

During the deployment pilot study at Axis we found that configuration of some analyzers differed between projects. Members from different teams at Axis requested that the MEAN system shall support native configuration of analyzers, by reading a configuration file in the project (FR2 in Section 6.2.7). This would definitely make the transition for a team to start using the MEAN system quicker, but which effects would it have on the system? The local MEAN-configuration will have to specify where the project analyzer-configuration file is located, if not every project uses the same location. The analyzer container protocol will have to support reading and merging an analyzer-configuration with existing analyzer-configuration in the container.

If the approach of supporting native configuration of analyzers is taken all parameters that can be configured for an analyzer are available on a project level. If taking the approach of limiting configuration to the (possibly expanded) common configuration protocol, all tools share the same interface and MEAN analyzer container contributors will not have to implement any configuration merging logic. Different releases of MEAN analyzer containers can cover use cases not covered by analyzer generic configuration, e.g. if indentation is not covered by the generic configuration, *Pylint-tabs* and *Pylint-spaces*, could cover two use-cases. The configuration protocol could be expanded with both support for native configuration and new analyzer generic parameters. This leaves the decision to the users by keeping both approaches possible.

7.4 Which is the Best Integration into Code Review?

In this section we discuss the integration of data-driven program analysis into code review. This includes what has been successful and what can be improved.

7.4.1 Which Findings to Present to the User

There is no point in presenting findings that are not useful in the context where they are presented. As MEAN was integrated into code review during the deployment at Axis, all analyzer findings in changed files were posted as robot comments. Some users requested that only findings related to changed lines would be posted as robot comments. The argument for this is that defects unrelated to the current change has to be fixed in another change for

changes to stay coherent. This means that comments on lines unchanged by a patchset are not actionable in the context where they are presented.

Simply filtering the robot comments on changed lines, have the potential of missing new defects introduced by a change, even for less complex defects. An example of a defect that would be missed is illustrated by changing code 7.1 to code 7.2. The variable `term` will be reported as undefined on line 2 after the change, while the only changed line is line 1.

Code 7.1: Create function change

```
1 def add_one(term):
2     return term + 1
```

Code 7.2: Remove parameter change

```
1 def add_one():
2     return term + 1
```

A sensible way to get around this problem is to post robot comments on defects introduced by a change. *Coverity* had the feature of only finding new defects in relation to a reference build, but most analyzers do not have this feature. A way to report only new defects is to run an analysis on the parent of a change to be used as a reference. The output of the reference analysis can then be compared with output from newer analyses. Defects reported for the same line of code with the same category present in the reference analysis can then be deleted from result of the analysis of the changed code. Note here that line number is not the same thing as line of code, because lines can be added and deleted in a change. The line numbers of the parent change can be mapped to the line numbers of the change with knowledge of which rows were added and deleted. For example changing Code 7.1 to Code 7.3, would mean that line number 2 in the parent change is mapped to line number 3 in the change that adds the documentation string. If Code 7.3 had Code 7.1 as parent a defect would be reported, but with 7.2 as parent no defect would be reported. This approach assumes that there is not several defects with the same category on the same line. If there is, we need another value to differentiate between the defects e.g. column number.

Code 7.3: Add documentation change

```
1 def add_one():
2     """ Returns term plus one """
3     return term + 1
```

For compiler based analyzers new errors often show up outside of changed files, even in another software repository. It is discussable how to show these errors in code review, because the only files that are easy to view in a Gerrit change are changed files. If the defect found outside of the changed files can be traced to a certain changed line, it would be a great option to present the defect inline at that position with information about where the defect showed up. If the defect cannot be traced, a simple message with information about where the defect showed up or an inline view of the defect outside of the change are valid options.

If posting robot comments for entire changed files, a Gerrit feature to toggle visibility of robot comments on unchanged lines would make it easier for the users to find robot comments introduced by a change. A more complex Gerrit feature to filter the diff of robot

comments between *base* change and a patchset would address the risk of missing new defects on unchanged lines.

7.4.2 Reducing Flooding

Only presenting defects new for a change reduce the flooding of developers. Still big changes have a risk of flooding. Another way to reduce flooding is to limit the number of alerts of a certain category presented for a file. For example if an analyzer finds 20 defects of the same category in the same file, three of them can be presented to the user with an additional comment that tells the user that there exist more findings in this category. This could be combined with a code review feature to toggle between that all comments are shown and only a maximum of three of each category are shown. If the view limiting the amount of comments is the default view, a comment regarding that more findings in the same category exist would help the user missing any comments.

7.4.3 Suggesting and Applying Fixes

Some analyzers report findings together with an applicable fix. Fixes provide valuable additional information about a finding [25]. The usage of *Tricorder* at Google tells that users often looked at suggested fixes in code review, but then fixed them in their editor while fixing other review comments [25].

No analyzer used during the deployment at Axis provided fixes, but viewing and applying fixes can be integrated into Gerrit. Fixes applied in Gerrit create a new patchset. A new patchset often lead to reviewers getting notified. A solution, where fixes applied in code review and local edits can be uploaded as one patchset, could lead to more fixes applied in code review.

7.4.4 Configuration

The project level configuration with inheritance, used to configure MEAN-analyzers, was a successful integration into Gerrit. A project is a reasonable smallest unit to agree on a configuration and the project tree structure in Gerrit (where parent projects only contain configurations) make it easy not to duplicate configuration.

During the pilot of MEAN we changed the configuration based on feedback from the whole R&D. This approach may not be optimal considering the different departments or teams may not have the same needs. A situation where a specific check is disabled because it gives many false positives for only one team prevent other teams from getting true positives from this check is not good. On the other hand posting comments that is almost always false positive is not good either. That each team take responsibility for how to configure the analyzer could be a way to go. In our solution you could only configure analyzers to disable checks, but many analyzers support other stuff that you can configure, that a user of our system would like to be able to configure. For example a problem that we encountered was that shell script many times started with a shebang that suggested that the script was an ordinary shell script (#! /bin/sh), but this path was a symlink to another like bash or dash. This lead to extra comments that was not useful and possibly loss of other comments that

would be shown if Shellcheck knew what kind of shell it was. In this case it would be useful if you could configure Shellcheck to analyze the file as a specific shell. Which is supported by Shellcheck but not our system.

7.4.5 Monitoring User Feedback

During the time the MEAN system analyzed code at Axis, we monitored the feedback and disabled the analyzers. Much of the information we gathered was also available to many other developers at Axis, but we did not advertise it. The ability to disable analyzers and categories was available for the developers. Having a few developers that do configuration of all analyzers for all teams may not be optimal in every case. There is merit to having an admin that are able to do some global configuration. If there is an analysis that always should run on every project an admin may want to force it globally or if an analyzer is not working correctly an admin may want to block it. But for analyzers and categories that may not work globally an approach were each team configure their own analyzers and categories themselves may be more beneficial. Having the whole team configure the analyzers together would be preferred over one developer to prevent biased configuration. Our implementation of MEAN at Axis cater to this style of handling the configuration by having the local configuration stored on Gerrit. When a local configuration change is uploaded to Gerrit other team members are able to see the change. A way how this could work is that a team member see that a category has been rated as not-useful many times, the member upload a change that disable this category. Now other team members have the ability to agree or disagree about the change. In the *Tricorder* paper [25] they talk about analyzers written by developers at Google and that the analyzer writer is responsible for the analyzer. If reported bugs filed against the analyzer is not fixed or that the analyzer is too annoying it will be disabled. A similar approach could be used at Axis for custom analyzers. For third party analyzers a developer familiar with the analyzer could be responsible.

Artificial intelligence (AI) could be used to automate the monitoring of user-feedback. The AI could also configure the system. A first step could be to add sensors that disable a category when it's not-useful-rate is above 5%. This can be done on global or project level.

7.4.6 Monitoring Running Analyzers

When the MEAN system analyzed code, the different stages of an analysis was logged. This made it easier to find bugs or other anomalies when we designed our system. Having a way to monitor the running analyzers for admins are essential to be able to find out if analyzer is misbehaving, e.g. taking too much resources. When the system was running at Axis we had an overview of the running analyzers from Jenkins and our different micro-services that printed info on the terminal, but there was no way to see this from Gerrit. To have a way for developers to see the status of currently running analyzers in Gerrit where they work could be good. Especially in cases where the analysis take longer time. Otherwise the developers might think the analyzer is not working or that no issues has been found.

7.5 What Can Be Shared?

The MEAN ecosystem is intended for a community where collected data, analyzer containers and other system components and can be shared. Stakeholders can together make better decisions and reduce setup time overhead.

7.5.1 Sharing Data

Collecting actionable data can take time if collected by a small group. By involving a bigger group of program analysis users and creators better decisions can be taken faster. The data about what is useful or not could be used within the company to decide beforehand what type of configuration an analyzer should have when it is introduced for a new project/team. This data could also be shared outside of the company to a wider community. Many analyzers are open-source and is not usually connected to a specific company. This means that developers of analyzers may have limited information on what types of checks work in a company and what type of checks do not work.

7.5.2 Sharing Analyzers

Many teams and companies use the same program analyzers. Setting up and integrating an analyzer, and manually configure it until it has the desired behaviour may take some time. With the MEAN system you can share analyzers and configuration between different teams to prevent duplicated work. If companies share analyzers and analyzer configurations between them further reduction of duplicated work could be achieved.

A big set of analyzers following the same protocol makes the process of work to integrate them into other systems a lot faster. Even if the MEAN Analyzer Container protocol, static analysis exchange format introduced by Kern et al. [22], the Shipshape analyzer protocol [13] or any other protocol is used, the effect of the tool agnosticism is faster integration.

7.5.3 Sharing MEAN Components

To make it more manageable for a big number of independent companies to use the same data-driven program analysis platform, the platform need to be flexible and cover a lot of use cases. The CI infrastructure of a company is rarely exactly the same as another company. Because different companies have different needs, it is not possible to create a CI-solution that work for every company. But many companies usually use third-party programs in their CI-infrastructure, and these programs are usually used by several companies. MEAN is possible to integrate with a simple load balancer and more complete CI solutions. The modularity of MEAN do not only make it possible to integrate the system with different CI-programs, it also makes it possible to share parts of the system between different companies.

MEAN is also designed to have interchangeable code review and data storage solution. Adapter services that integrate with code review and data storage can possibly be shared between companies. If a company has created a publisher that integrate with Gitlab, they could share it with other companies.

Finally, the communication infrastructure does not look the same at every company and companies may not use the same communication protocol. The MEAN main service is a framework, which can be be used with third party components that handle the communication. These components are reusable for companies using the same communication protocol.

Chapter 8
Threats to Validity

In this chapter we make sure information and results from data-analysis, surveys and interviews are interpreted for what they are, by addressing known threats to validity.

8.1 Data-analysis

The metric used to count fixed robot comments is the sum of differences in number of published robot comments between the first and the last patchset. This can also be explained by removing files or chunks of code from a change, without fixing the defects in robot comments. There is a risk that the number of fixed robot comments are estimated higher than the number of robot comments that were actively fixed.

Some developers at Axis used Gerrit with an old user interface or a command line interface. These two interfaces did not support displaying and responding to inline robot comments. This may explain some of the cases where robot comments were not responded to.

One user could not report the same robot comment in the same patchset as not-useful more than once. If the robot comment reappeared in many patchsets, all duplicates could be reported as not-useful. This means that some users did report not-useful for the same comment in patchset after patchset and some users did not. The not-useful rate is as low as 2.5% if all users did repeatedly reported not-useful and about 5% if all users did not.

When counting unique defects, the properties used were: category, line number, file, and repository. When code is changed, a defect can move to another line while still being the same defect. This means that the same defect was counted again when it moved to a new line. This metric of unique defects is an upper bound of unique defects.

As seen in the deployment timeline in Table 6.1, the analysis of comments on changed lines was done with an addition of data collected for two weeks. During this period some new robot comments were published on changed lines to the Streaming- and Tools-projects. Also responses to old robot comments were collected during these two weeks. It is probable that this had a small effect on the comparison between responses to robot comments on changed

and unchanged lines.

8.2 User Surveys

The names of survey participants were collected with their answers. The results of the survey were likely affected by the social context where questions were asked. It is possible that a *participate response bias* [19] made survey participants answer questions to please us as receivers of the answers. That is by being more positive about the robot comments they perceived than they actually were. When selecting a sample of users, there is a risk of the sample not being representative.

8.3 Interviews with Senior Employees

The interviews with senior Axis employees were conducted to put the deployment of MEAN at Axis into a context. The nature of the interviews were exploratory. With a sample size of three persons, there is a risk the sample is not representative. All three employees talked from knowledge and experience within their field (security, video streaming & Linux kernel). It is not sure that the examples taken up during the interviews are applicable to other fields at Axis.

Chapter 9

Conclusions

In this master thesis we created a data-driven program analysis system named MEAN, integrated it into code review and deployed it at Axis. The MEAN system was created with a community in mind, where users can contribute with data, analyzers and other system components. We defined tool-stack-agnostic protocols that were possible to use on the Axis tool stack, but may have to be tested on other tool stacks before reaching a stable state. Before analyzers are contributed, we recommend the MEAN container protocol to be extended with an option for the MEAN container to communicate its prerequisites.

MEAN was deployed at Axis for 7 weeks, of which 2 weeks for the whole R&D department. During these 7 weeks MEAN analyzed 485 changes, which were uploaded or reviewed by 407 users. Over 20 000 analyzer findings were presented in Gerrit code review as inline robot comments. The users could respond to these comments by clicking a "NOT-USEFUL"-button, which they did once per 40 published comments. The "NOT-USEFUL"-clicks helped us making data-driven improvements by disabling 16 categories of analyzer findings. These categories of findings were disabled because they were false positives, too picky or flooding the users. During the deployment all analyzer findings in changed files were published as robot comments. After the deployment we conducted a user survey. The answers led us to hypothesize that findings on unchanged lines were to be treated as noise in code review. An estimation of fixed defects showed that one out of ten defects found on all lines were fixed, while one out of three defects found on changed lines were fixed. The user feedback and higher fix rate on changed lines, shows that program analysis integrated into code review shall only present the user with findings introduced by the current change.

9.1 Future Work

The long term goal of writing this thesis is to adapt program analysis to the needs of the users, specific to the context where analysis is used. Future work can follow several tracks, with focus on areas such as data collection, artificial intelligence and workflow integration. We have conducted a list of ideas that can inspire next steps:

- Automate the data-driven configuration of analyzers by adding sensors, which trigger configuration.
- Deploy MEAN for a longer period where only defects introduced by a change are presented.
- Investigate generic approaches of running diff analysis, which only presents defects introduced by a change.
- Run data-driven analysis with new points of integration.
- Collect new data from users reacting to alerts. This could include everything from adding new buttons to tracking eye-movements.
- Apply deeper statistical analysis, to make better decisions when configuring program analyzers.
- Add execution of unit tests to the data driven loop.
- Investigate which data companies are willing to share.
- Investigate further how program analysis tools, without losing functionality, can be generic to their environment, e.g. by extending the *MEAN container* protocol.

Bibliography

[1] Axis – history. https://www.axis.com/about-axis/history. Accessed: 2019-12-13.

[2] Axis – homepage. https://www.axis.com/. Accessed: 2019-12-13.

[3] Coverity scan - home. https://scan.coverity.com/. Accessed: 2020-02-03.

[4] Cppcheck - manual. http://cppcheck.sourceforge.net/manual.pdf. Accessed: 2020-02-03.

[5] Docker – homepage. https://www.docker.com. Accessed: 2020-01-17.

[6] Elastic. https://www.elastic.co/. Accessed: 2020-03-21.

[7] Gcc - home. https://gcc.gnu.org/. Accessed: 2020-02-03.

[8] Gerrit review labels. https://gerrit-review.googlesource.com/Documentation/config-labels.html. Accessed: 2020-03-23.

[9] Git - home. https://git-scm.com. Accessed: 2020-02-03.

[10] Kythe - home. https://kythe.io. Accessed: 2020-02-03.

[11] Python - home. https://www.python.org/. Accessed: 2020-02-03.

[12] Remote procedure call - wikipedia. https://en.wikipedia.org/wiki/Remote_procedure_call. Accessed: 2020-02-03.

[13] Shipshape, Github page. https://github.com/google/shipshape. Accessed: 2019-10-15.

[14] Sparse - docs. https://www.kernel.org/doc/html/v4.12/dev-tools/sparse.html. Accessed: 2020-02-03.

[15] Tricium - readme. https://chromium.googlesource.com/infra/infra/+/master/go/src/infra/tricium/README.md. Accessed: 2020-02-03.

[16] Tricium plugin for gerrit. `https://chromium.googlesource.com/infra/gerrit-plugins/tricium/`. Accessed: 2020-03-21.

[17] Valgrind - home. `https://valgrind.org/`. Accessed: 2020-02-03.

[18] What is a container? `https://www.docker.com/resources/what-container`. Accessed: 2020-01-17.

[19] N. Dell, V. Vaidyanathan, I. Medhi, E. Cutrell, and W. Thies. "yours is better!": Participant response bias in hci. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, page 1321–1330, 2012.

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

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

[22] M. Kern, F. Erata, M. Iser, C. Sinz, F. Loiret, S. Otten, and E. Sax. Integrating static code analysis toolchains. *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), Computer Software and Applications Conference (COMPSAC), 2019 IEEE 43rd Annual, COMPSAC*, page 523, 2019.

[23] M. Monperrus and M. Mezini. Detecting missing method calls as violations of the majority rule. *ACM Transactions on Software Engineering and Methodology*, 22(1):1–25, 2013.

[24] M.G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran. Making defect-finding tools work for you. *2010 ACM/IEEE 32nd International Conference on Software Engineering, Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, 2:99 – 108, 2010.

[25] C. Sadowski, J. van Gogh, C. Jaspan, E. Soderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, page 598, 2015.

[26] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli. Modern code review: a case study at google. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 181–190, 2018.

[27] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall. Context is king: The developer perspective on the usage of static analysis tools. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 38–49, 2018.

[28] C. Wohlin, M. C. Ohlsson, M. Höst, P. Runeson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering.* Springer Berlin Heidelberg, 2012.

Appendices

Appendix A
GStreamer Checkers

Category	Description
C0001	Comments should use multi-line comment syntax.
C0002	Comment should contain some text.
C0003	Comments that start with "/*" and not "/**" start with a lowercase letter.
C0004	Comments that start with "/*" and not "/**" end with no period.

Table A.1: Categories of GStreamer comment checker.

Category	Description
C0001	Strings passed to a macro start with a lowercase letter for Uppercase macros, see Table A.3.
C0002	Strings passed to a macro end with no period for Uppercase macros, see Table A.3.
C0003	Strings passed to a macro start with a uppercase letter for Lowercase macros, see Table A.3.
C0004	Strings passed to a macro end with a period for Lowercase macros, see Table A.3.

Table A.2: Categories of GStreamer macro checker.

Uppercase macro	Lowercase macro
GST_ERROR	syserror
GST_WARNING	error
GST_INFO	warning
GST_DEBUG	info
GST_LOG	g_error
GST_FIXME	g_message
GST_TRACE	g_critical
GST_ERROR_OBJECT	g_warning
GST_WARNING_OBJECT	g_info
GST_INFO_OBJECT	g_debug
GST_DEBUG_OBJECT	
GST_LOG_OBJECT	
GST_FIXME_OBJECT	
GST_TRACE_OBJECT	

Table A.3: List of two different types of GStreamer macros.

Appendix B
Interview Protocol

This is an interview held as a part of the master thesis project "Data-driven Program Analysis Deployment".

In this study "program analysis" is used as a word for describing static analysis, dynamic analysis, linters and other code checkers.

In this study a few analyzers are deployed and data is collected regarding:

1. Which, by the analyzers, reported issues are not useful
2. Why are issues reported as not useful
3. Which reported issues get resolved

The purpose is to make analyzer results more useful.

1. What is your background at Axis?
    (a) How many years?
    (b) Which parts of organization?
    (c) Do you have any other background that is relevant in the context of program analysis?
2. Which analyzers have been used at Axis? (if many, some important ones)
    (a) How has this changed over time?
    (b) What has been the user response (pain points)?
    (c) Have any analyzer been mandatory?
3. Which points of integration of program analyzers have been used at Axis?
    (a) How have analyzers been triggered?

   (b) Where have results been published?
   (c) What has been the user response (pain points)?

4. How has the integration and configuration been organized?
   (a) Has it been centralized or decentralized, e.g. at department, project or user level?
   (b) How has this changed over time?
   (c) How would you evaluate these solutions?

5. Has user feedback to analyzer results been collected?
   (a) If yes, how has this been organized?
   (b) If not, have there been informal or less organized feedback?
   (c) Have any actions been taken considering the feedback?

Appendix C
User Email Survey

```
Hello,

You are one of 20 randomly selected developers that received Gerrit inline
comments from the user svcmean. We would be really thankful if you could take
part of a short survey by answering this mail.

The survey is a part of the master thesis project
"Data-driven Program Analysis Deployment".The intention is to evaluate the
deployment of the MEAN data-driven program analysis system.

Conditions:
- I understand that my name will be anonymized in the master thesis.
- I am aware that I can contact the master thesis authors
  (anton.ljungberg@axis.com & david.akerman@axis.com) at any time to access
  further information.

---

Questions:

1) How many years have you worked with software development?


Pick one of your changes that received comments from svcmean.
A list of such changes can be found at:

Question 2 & 3 are asked regarding this change.
```

```
2) What is the change id of the change you picked?

3) How do you perceive the comments from svcmean in this change?


Go to the first patchset in the change with comments from svcmean.
Question 4, 5 & 6 are asked regarding this patchset.

4) How many of these comments do you agree with?
   You may neither agree or disagree with some comments.

5) How many of these comments do you disagree with?
   You may neither agree or disagree with some comments.

6) If you would click "NOT-USEFUL" on any comment, why would you do so?


For question 7 & 8, take all comments you received from svcmean into consideration.

7) Which were the main reasons for comments being not useful?

8) What would you like to change regarding comments from svcmean?


---

Thank you for contributing to more useful program analysis!

Best Regards,
Anton Ljungberg & David Åkerman
```