Case Study on Data-driven Deployment of Program Analysis on an Open Tools Stack

Ljungberg, Anton; Åkerman, David; Söderberg, Emma; Sten, Jon; Lundh, Gustaf; Church, Luke

Published in:
Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice

DOI:
10.1109/ICSE-SEIP52600.2021.00030

2021

Case Study on Data-driven Deployment of Program Analysis on an Open Tools Stack

Anton Ljungberg
Qlik
Lund, Sweden
anton.ljungberg@qlik.com

David Åkerman
Axis Communications
Lund, Sweden
david.akerman@axis.com

Emma Söderberg
Lund University
Lund, Sweden
emma.soderberg@cs.lth.se

Gustaf Lundh
Axis Communications
Lund, Sweden
gustaf.lundh@axis.com

Jon Sten
Axis Communications
Lund, Sweden
jon.sten@axis.com

Luke Church
University of Cambridge
Cambridge, United Kingdom
luke@church.name

***Abstract*—**
Program analysis can assist software development but its use is limited by a number of usability issues, including false positives and integration issues. Previous systems have show significant progress addressing these by using a data-driven approach to track the 'not useful' results, and integrating the results into code review. However, these systems were very context dependent. In this paper, we motivate the application of the same data-driven principles and code-review based integration to the design of the MEAN system, an analysis infrastructure running on an open tools stack. We then evaluate its implementation within a local software company, and present implications for the design of future program analysis systems.

***Index Terms*—program analysis, usability, case study**

I. Introduction

Program analysis tools can provide useful information to help software developers improve performance, make code more maintainable, fix bugs and perform many other tasks. However, the use of program analysis results is hindered by a number of usability issues [1]–[3]. These include distracting false positives, incomprehensible messages, and poor integration into developer's workflows.

One approach to tackling these issues is that taken at Google by the Tricorder system [4], a static analysis platform that integrates program analysis results into the developer workflow at the point of code review. Here, program analysis results are displayed as generated comments. The comments have buttons associated with them where developers can indicate that the comment was 'not useful'. Engineers tune the infrastructure to maintain a so called 'not useful' rate of below 10%. The tuning of the system is by a large extent done by analyzer maintainers outside the Tricorder team.

The Tricorder system exemplifies Google's philosophy for program analysis [4] expressed as the following five principles; 1) "no" false positives, 2) empower users to contribute, 3) make data-driven improvements, 4) workflow integration is key, and 5) project customization over user customization. Principles 2 and 5 address challenges beyond the individual usability issues described above, for instance, how to handle the maintenance load of many analyzers, and how to handle disagreement about analyzer results. Whilst these issues don't limit the usability of the analysis results on an individual basis they become significant issues when a platform is broadly adopted.

The results that this philosophy has attained are encouraging for the adoption of program analysis, however it is not obvious that this success generalises outside Google. In this work we explore whether it is possible to reproduce the success at Google in another context. The Tricorder deployment at Google has a number of particularities. It is developed and deployed within a single large corporation, so it is comparatively easy to justify the allocation of engineering resources to build infrastructure that benefits the engineers across the company and to build a sense of community where engineers using the system take the time to submit feedback in the confidence that it will be acted upon.

Secondly, the code base at Google is stored in a single monolithic repository with a standardised set of working practices around it. Whilst this has resulted in the need to develop special purpose tools, it has created a single point of integration within the developer workflow. This is significantly different to common multi-tool, multi-repository setups used in other organisations.

In this paper, we describe the design and deployment of an open source system called MEAN (for MEta-ANalyzer system), that embodies the five Tricorder principles. With no existing open platforms embodying the philosophy above[1], we conducted an experiment where we developed an open platform in close collaboration with the developer infrastructure team at the company, which we then gradually deployed.

This work was partly funded by the Swedish Foundation for Strategic Research (grant nbr. FFL18-0231), the Swedish Research Council (grant nbr. 2019-05658), and Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

[1]The closest system is the Tricium system [5] run by the Chromium project (also supported by Google) but it closely tied to custom infrastructure used by the Chromium project.

As well as the open nature of the platform, we expand upon previous work by developing better indicators of the success of the approach. The Tricorder system reports users clicking on results on average 0.8% of the time, and 5% out of the times users clicked on results they reported that comments were 'not useful'. Whilst these metrics indicate significant usage when applied to the scale of a Google codebase they leave open questions - how many of the results are being used? How were they perceived by developers? What would make the system more approachable? By designing a socio-technical method for the design, deployment and operation of an analysis system, we explore the possibility of a *data-driven program analysis deployment*. In doing so, this paper makes the following contributions:

- The open-source MEAN system.
- A description of how developers responded to the system during their work.
- Design recommendations for static analysis infrastructure to support adoption.

The rest of this paper is structured as follows; we give some background and cover related work in Section II, describe our methodology in Section III, before we introduce the results of our study in Section IV, Section V and Section VI. We identify threats to validity in Section VII, and then we discuss results in Section VIII and conclude in Section IX.

II. Background and Related Work

In this section, we briefly describe the Tricorder system and cover related work exploring data-driven improvements to program analysis usability.

A. The Tricorder System

The *Tricorder* system [4] is a closed source program analysis platform developed at Google for its internal developer infrastructure. The system has a microservice architecture in the form of a static analysis pipeline that reacts to events in the developer workflow. The pipeline computes analyzer results at different stages; FILES, DEPS, COMPILATION. At each stage information is added, starting with the content of a file, for lexical analysis, and ending with dependency information for replaying of a compilation for compiler-based analysis. The primary workflow integration point is code review where the system reacts to uploads to so called changelists under review in the internal code review tool.

When results are reported as so called robot comments in code review, users can interact with robot comments by clicking on buttons on the comment. Users have the option of, for instance, clicking Please Fix, Apply Fix, or Not Useful on a robot comment. Please Fix is available on all robot comments and provides the means for users to communicate around a robot comment in a code review. The Apply Fix button is available on robot comments offering a suggested fix, which is applied if users click on the button (they can also preview solutions in this case). If a user clicks the Not Useful button they get the option of reporting a bug with more details about why the comment was found not useful. The additional information is provided in a bug tracking tool, which users get redirected to via a link. All clicks on comments are logged and used to compute a so called Not Useful rate (Not Useful/ Not Useful + Please Fix + Apply Fix), which can be broken down to specific analyzers and their different categories. This rate is used to monitor and tune the system to maintain a rate below 10%.

During 2014, the system produced on average 93K results per day and on average 716 Please Fix and 48 Not Useful clicks daily. Based on these numbers, there was an estimated average user engagement with comment of 0.8% on a daily basis. The overall Not Useful rate of the system was around 5% during 2014, while the Not Useful rate of individual analyzers and their categories was occasionally higher. Any analyzer getting a rate above the threshold of 10% is put on a probation list where analyzer maintainers are engaged to address the issue, and if any analyzer category receives a rate above 25% it is turned off directly by the system maintainers.

Further, there is a reported learning effect where developers learn from seeing antipatterns flagged in code review. The effect is manifested as users stopping to introduce certain problems into the code base at the same time as they start to fix existing occurrences. The Tricorder system is also designed for plugability, and this property appears to have been successful with many contributions of analyzer from outside the analyzer team (13 of 16 reported analyzers are contributed from outside the maintainer team).

B. Data-driven Improvements to Program Analysis Usability

The Tricorder system has two descendants developed in close proximity to the system. The first system, *Shipshape* [6], was developed shortly after the report on the Tricorder paper was published as an open-source version of Tricorder. The Shipshape system utilizes a similar microservice architecture as that used in Tricorder, but adapted to an open-source setting with fewer dependencies to the details strongly coupled to the internal Google infrastructure. As a consequence, the system has fewer stages (PRE-BUILD and POST-BUILD, mapping to the FILES and COMPILATION stage in the Tricorder system) and is designed around use of Docker containers, where analyzers are bundled as Docker images that implement a remote procedure call API. The Shipshape design enables pluggability, but there is no support for gathering of 'not useful' feedback, and the system has since been archived.

The second system, *Tricium* [5] is developed for the open-source Chromium project hosted by Google. The Tricium system lifts the concept of stages into a configuration language, detaching it from the system design and making it easy to add new stages if needed without changing the system. It is further open-source, designed for pluggability and it integrates with code review via the Gerrit code review system. However, its pluggability approach and its execution platform is closely coupled to the specific infrastructure of the Chromium project which limits the possiblity for adoption in other contexts.

Considering approaches further away from Tricorder, Nanda et al. [7] deployed a static analysis online portal called

Khasinia at IBM in 2010. The main design goal of Khasinia was to provide static analysis as a service, using a portal as a way to skip client tool integration and eliminate installation overhead. Users could upload code to the portal where it would be analyzed by a selection of tools. When done, the portal would list found defects where users could report defects as ”Invalid”, ”WontFix” ”Confirmed” and ”Not Attended”. Results from the used tools are merged and aggregated, then filtered and sorted based on user-specific analysis and user feedback. 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.

III. Methodology

In this section we present our research questions, research method and threats to validity.

A. Research Questions

The overall goal of this research is to explore the applicability of Google’s philosophy for program analysis to a different setting. We picked a company with a different culture and internal organisation and based on a more typical open tools stack. We seek to answer the following research questions in this research setting:

RQ$_1$ What is the experience of using program analysis? To inform the design and deployment of a program analysis infrastructure we need to first characterise what the existing experience developers have with program analysis is.

RQ$_2$ What does a system based around an open tools stack look like? It is not clear to what extent Google’s philosophy for program analysis is shaped by its approach to managing code and development practices in general. In order to explore this, we build a similar system, but based around an open tooling stack, and consider the design decisions involved, and what they imply about the feasibility of applying the philosophy outside Google.

RQ$_3$ What is the effect of deploying the resulting system? In the exploration of RQ2 we unpack the technical choices that need to be different in an open tools stack. However for these to be relevant the system must be beneficial. We expand on the limited evidence for the effects of deploying Tricorder, by answering the following:

- **RQ$_{3.1}$** What is the level of user engagement?
- **RQ$_{3.2}$** How many results are used?
- **RQ$_{3.3}$** What is the user experience?
- **RQ$_{3.4}$** What is the maintainer experience?

B. Research Method

To answer our research questions we used case study research [8] in an exploratory fashion. The case study was performed at a company with over $3,000$ employees of which $1,150$ work in R&D on software and hardware development. The teams within R&D have broad control over their environment, and a number of editors and IDEs are used. Open developer tools like Git[2], Gerrit[3], and Jenkins[4] (an open tool stack also reported from other companies [9]), are used by a large portion of the developers, and the company has a team dedicated to maintaining these tools. During the experiment described here, the Gerrit code review system had more than 850 unique active users per week.

We gathered data from different sources; semi-structured interviews with senior employees connected to historic use of program analysis at the company (**RQ$_1$**), informal discussions with the maintainers of the developer infrastructure during seven software development iterations to inform the system design (**RQ$_2$**), quantitative logs analysis from a deployment of the system over 8 weeks together with a follow-up survey to get insights into the effects of deploying the system in this context (**RQ$_3$**).

Interviews. We conducted 3 semi-structured face-to-face key informant interviews [10] with senior engineers at the company who were involved in the deployment of program analysis at the company. The interview subjects were selected by the authors of the paper asking their contacts who should be interviewed. The selected participants have been involved in creating a program analysis culture at the company in the fields of security, kernel development and video streaming development. They all had over 10 years of experience in their respective fields, and two of them had been working at the company for over 20 years. The interview protocol, included questions about the use of program analysis at the company, how it has been integrated and configured, and how user feedback has been managed. The purpose of the interviews was to help form the research hypothesises and ensure that the experiment was aligned to the practice at the company. The interviews were recorded and then reviewed by the authors for key themes and insights that are described in this article.

System Development and Deployment. The MEAN system was designed by a comparative survey of the architecture of the existing systems (Tricorder, Shipshape, Tricium), followed by 7 weekly iteration cycles with design meetings with the maintainers of the developer infrastructure at the company. Subsequently we deployed the MEAN system to more users over a period of 8 weeks, starting with 3 weeks for the developer infrastructure team at the company (35 users), then an identified client team was added for an additional 3 weeks (adding 50 more users), and then finally, all of the R&D team was added for the final 2 weeks. The initial deployment, and the deployment to all of R&D, was proceeded by a ”quiet” roll out where results were computed but not presented[5]. This progressive roll-out enabled the developers of the system to ensure that its behaviour was reasonable before affecting a large number of engineers. Based on informal discussions with the developer infrastructure team and the client team about their program analyzer needs, we selected 7 analyz-

[2]Git - a distributed verson control system, https://git-scm.com/

[3]The Gerrit code review tool, https://www.gerritcodereview.com/

[4]Jenkins - a continuous integration system, https://www.jenkins.io/

[5]Any time we report number of results it refers to results published as robot comments, i.e., not results computed in quiet deployments.

ers to integrate into MEAN; Ansible-lint[6]. CommentCheck[7], Flake8[8], Hadolint[9], Macrocheck[10], PyLint[11], and Shellcheck[12]. For each analyzer, we wrapped it in a Docker container complying with the MEAN analyzer API.

During the deployment we monitored the published robot comments and NOT USEFUL clicks, for every analyzer category. We used a 5% NOT USEFUL rule to guide decisions, where a category was disabled if more than 5% of the published comments within that category were reported as not useful. The reliability of the percentage of NOT USEFUL robot comments for a category was judged to be high enough to disable a category if there were 100 existing robot comments in that category. In effect, this meant that categories with less than 100 published robot comments and at least 5 NOT USEFUL clicks were disabled. The 5%-rule was complemented with manual inspection of robot comments and informal user feedback regarding robot comments. We paid extra attention to the manual inspection during the "quiet" roll out of newly introduced analyzers, and in the early stages of deployment we paid extra attention to informal feedback.

Analysis of Robot Comment Data. In line with Tricorder, we call comments that contributed to a review by an analyser 'robot comments'. We analyzed these comments to estimate the number of results from MEAN that were used. We define a comment as showing one result, and by 'used' we mean that the code was changed in such a way to prevent the comment occurring. When we refer to a result as being "fixed" we mean its corresponding published robot comment to have been used. We estimate this by computing the number of robot comments in the last patchset minus the number in the first patchset, taking into account cases where the change could be caused by configuration changes rather than developer actions. We computed used comments using unique robot comments in a change, and we computed unique comments by removing duplicates. We consider two robot comment to be equal (i.e., duplicates) if they have the same category and appear on the same line and file. In addition, when we report NOT USEFUL clicks, we only count a maximum of one click per user for each robot comment.

Survey. We created a questionnaire in the form of an email that was sent to 20 randomly selected MEAN users. Half of the selected users were owners of changes with used robot comments, and half without. We composed a list for each survey recipient with changes that they owned which had received robot comments. They were then asked to pick a change from the list and to answer questions in relation to this change. With this strategy, we aimed to mitigate the risk of recall bias. The survey was answered by 10 users, their responses where open-coded to observe any patterns in why 'not-useful' was clicked. Whilst this small sample can't be considered conclusive the answer give indications about common patterns reported in the results section.

[6]Playbook analyzer, https://ansible-lint.readthedocs.io/

[7]Coding convention check GStreamer, https://gstreamer.freedesktop.org/

[8]Python style checker, https://flake8.pycqa.org/

[9]Docker file analyzerhttps://hadolint.github.io/hadolint/

[10]Coding convention check GStreamer, https://gstreamer.freedesktop.org/

[11]Python file analyzer, https://pypi.org/project/pylint/

[12]Shell script analyzer, https://www.shellcheck.net/

IV. RESULTS: EXPERIENCE OF PROGRAM ANALYSIS

In order to understand how the design may need adapting to the context, we used key informant interviews to build an understanding of the use of program analysis at the company. The results that are relevant to (**RQ_1**) are described here. The interview subjects are referred to as (S1)-(S3).

To a large extent **teams have a lot of freedom** and can adjust their development practise to their needs. Consequently, there is variation in how developers use program analysis, it is used in editors, as pre-commit hooks in Git, and in the continuous integration system Jenkins. Jenkins has a connection to the code review system Gerrit, where it posts a message with a summary and a link to a Jenkins log with more details.

The **extent to which teams use program analysis varies**. (S3) reported that *"other methods to minimize the number of defects are at times more efficient"*, with some teams preferring other approaches to code quality, such as testing. The interviewees viewed the decision to start to use program analysis, especially static analysis, as a trade-off between costs and benefits, as viewed by management.

Finding 1. *The practise around use of program analysis varies between teams and is tied to a cost/benefit analysis on team level.*

The cost of using static analysis has been salient to developers. **Triaging of results to identify positives can be a very costly activity**. Triaging is often complex and requires domain-specific knowledge, and does not fit within developers' daily work practices. The company has explored an approach where a specialized team runs certain analyzers, triages the output, and then files bugs against different components. This strategy was found to be especially useful for components with less active development.

This experience with triaging has resulted in a practice where the tools used on a daily basis only integrate certain analyzers, referred to by one interviewee (S2) as analyzers *"built for zero false positives"*. Static Analyzers that have been found to be acceptable from this "zero false positives" perspective include Cppcheck [13], analyzers included in the gcc compiler [14], and analyzers focusing on a very specific context, like Sparse [15]. (S2) summarized the mindset as *"analyzers that do 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"*.

Another issue is that **analyzers flood users with too many results**. (S1) mentioned an example with an analyzer for C

[13]http://cppcheck.sourceforge.net

[14]https://gcc.gnu.org

[15]https://www.kernel.org/doc/html/v4.12/dev-tools/sparse.html

which reported so many results it hampered developers' work, resulting in developers across the company stopping usage of the analyser. All interviewees agreed that analyzers that find too many defects are not suitable for integration into daily used tools.

None of the interviewees were aware of any systematic approach for gathering developer feedback relating to program analysis.

Finding 2. *The cost of managing false positives have lead to a practice where use of analyzers in daily development is limited to analyzers that are "built for zero false positives". There has further been no systematic gathering of user feedback in relation to program analysis.*

The **experience of using dynamic analysis has been more positive**, for instance, Valgrind [16] is widely used at the company and is often used while running automated unit tests, however in some cases, especially with embedded systems, there may be memory restrictions preventing the use of dynamic analysis. In these places where dynamic analysis is not possible, using static analysis has been seen as a good investment.

Across the organisation, the company has been **moving towards a more centralized model** for the use of analysis. This is due to the costs associated with setting up analysis in a decentralised setting, resulting in complicated installation processes. One interviewee (S3) summarized the developer perception as *"developers at this point often want tools to just work in the build chain"*, while another interviewee (S1) summarized the shift in terms of the trade-off involved as *"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"*.

Finding 3. *A more centralized model for developer tool maintenance moves part of the cost/benefit analysis of using program analysis to a dedicated tools team.*

V. Results: System Design

To address our second research question (**RQ$_2$**), we designed and implemented a system we call MEAN (MEta ANalyzer), shown in Figure 1, using the iterative approach outlined in Section III. We provide an overview of the design here and refer to the open source implementation [17] for more details.

[16]https://valgrind.org/

[17]https://gitlab.com/lund-university/mean

System Components The design uses a micro-service architecture with a main service (`MEAN Main`) that coordinates communication between other services and keeps track of system state, a publisher service (`MEAN Trigger`) that detects and propagates change events, an analyzer executor service (`MEAN Executor`) manages the execution of analyzers, a robot comment publisher service (`MEAN Publisher`) integrating with the code review tool, and a storage publisher service (`MEAN Storage`) that manages storage, e.g., analyzer results and user feedback.

System Control Every event and request in the system travels through the `MEAN Main` service, which has the following responsibilities; 1) compute request-specific configurations by merging local and global configurations, 2) determine which analyzers to run and how they should be configured, 3) emit `MEAN Event`:s to trigger execution of analyzer and to report results from the `MEAN Executor`, 4) keep track of the current state of analyzers to describe what is currently happening in the system. `MEAN Main` keeps track of requests by use of states; `Scheduled`, `NotRelevant`, `Started`, `Error`, `Timeout`, and `Result`. The `NotRelevant` state is used for cases where it is not relevant to analyze some files, for instance, PyLint should not analyze Java files. Requests get assigned `Scheduled` after they have been sent to `MEAN Executor`, and then later get assigned `Started` and finally `Error`, `Timeout`, or `Result`, depending on how the execution went.

System Communication To make the system less dependent on different communication protocols we designed the system around four interfaces that handle the communication between the services. As a core service in the system, `MEAN Main` receives `MEAN Request`:s and `Analyzer Event`:s, and sends `Analyze Request`:s and `MEAN Event`:s (shown in Figure 1). In our implementation we 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 valid.

Analyzer Execution The `MEAN Executor` service is responsible for running program analyzers. For each `Analyze Request` it receives, it walks through the following steps: 1) Creates a file tree for communication with the analyzer to run (with directories called `code`, `input`, `output`, 2) Translates the request to a JSON file (`input/analyze_request.json`), 3) Updates `MEAN Main` on progress (sends a *Started* event), 4) Retrieves the code to analyze (saves it in `code`), 5) Runs the analyzer with access to the created file tree, and starts a timeout timer, 6) When the analyzer is done it inspects the result (checks `output` and runtime behavior) and communicate the result to `MEAN Main`. In our implementation we implemented MEAN-executor on top of Jenkins, but this dependency is isolated from the rest of the system and another continuous integration system could be used instead.

Analyzer Containers Analyzers used by MEAN run as

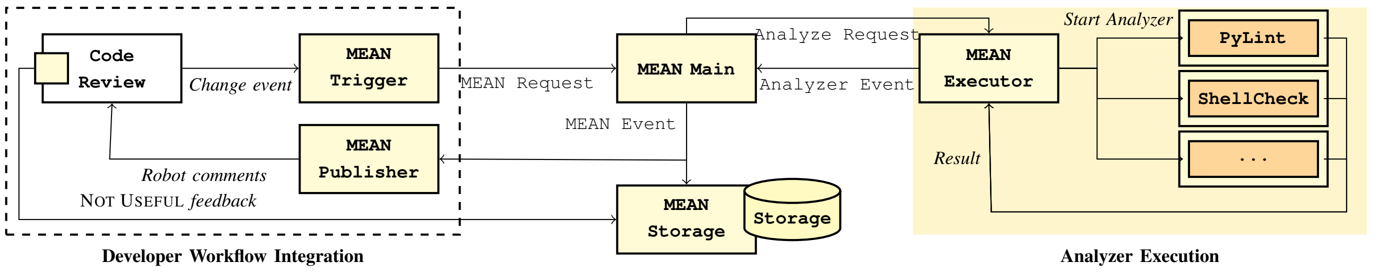


Fig. 1. **Overview of MEAN** An overview of the design of the MEAN system. `MEAN-*` map to services in the system. Solid arrows show events and requests in the system and dashed arrows show information being communicated. Yellow parts in the workflow integration section (two services and a code review plugin) and analyzer execution section (a service and Docker containers wrapping analyzer), indicate part connected to MEAN.

Docker containers[18], which are started each time a new request for an analyzer and a file is received. Consequently, to add an analyzer to the system, it needs to be wrapped in a Docker container and the input and output of the analyzer needs to be connected to how `MEAN Executor` communicates with analyzers via the earlier mentioned file tree. That is, analyzers can see the request in the `analyze_request.json` file, find files to analyze in the `code` directory, and communicate results via the `output/result.json` file.

Workflow Integration The MEAN system is triggered via instances of `MEAN Trigger` services that connect to different developer workflow integration points, for instance, an editor or a code review tool. Consequently, each integration point requires one dedicated `MEAN Trigger` service that translate events in the integration point to `MEAN Request`:s. Likewise, it needs to have a dedicated `MEAN Publisher` to integrate results.

Code Review Integration As the primary integration point for the Tricorder system, we aimed to also integrate MEAN with code review. To this end, we implemented a `MEAN Trigger` and `MEAN Publisher` to connect Gerrit, and we further implemented a Gerrit plugin to control the display of robot comments[19]. Using the options for interaction with human comments as a starting point, we let robot comments have the following buttons: PLEASE FIX, NOT USEFUL, DONE, and ACK. All interactions with a comment, except NOT USEFUL, can later be marked as RESOLVED, and we give both owners and reviewers the option of clicking NOT USEFUL. The possibility of giving further feedback is integrated into the interaction with comments, and any REPLY (or QUOTE) to a robot comment is collected by MEAN and summarized per analyzer and category.

Finding 4. *The cost of giving further feedback depends on the possibilities in the integration point. By letting users reply directly to comments in Gerrit we keep them in context, while in Tricorder users are sent to a different tool.*

[18]https://www.docker.com

[19]We used an existing Gerrit plugin developed for Tricium as a starting point, https://chromium.googlesource.com/infra/gerrit-plugins/tricium/.

We decided to publish robot comments on unchanged lines because a change may produce new robot comments on unchanged lines. Tricorder also publish comments on unchanged lines, but has the comments on unchanged lines hidden by default and leaves it up to the user to toggle the visibility of these comments [4]. The version of Gerrit we used did not support toggling comments on unchanged lines.

Configuration To support configuration differences between teams, analyzers are configured via a global and a local configuration. The global configuration is stored with the system, while the code review system is used to host the local configuration (we use the Gerrit plugin to manage this connection). All analyzers running in the MEAN system must have an entry in the global configuration, but these analyzers can be customized in a local configuration by a team. A configuration entry includes an analyzer name, analyzer status, Docker image, execution timeout, regex matching files to run on, and analyzer categories.

Feedback and System Tuning When designing the code review integration, we did not include the option of applying a fix. The reason for this was an uncertainty to what extent the Gerrit code review tool would support this kind of interaction without significant effort, and we also do not have analyzers in our selection that provide suggested fixes. One consequence of this decision is that no data for clicks of this kind are gathered in the system. Hence, we cannot include APPLY FIX when computing a NOT USEFUL rate, but we consider this to not be an issue as this is also the case for analyzers with no suggested fixes in the Tricorder system.

Still, this difference sheds light on the strong coupling of the interaction options and the integration point, and a more general description of the NOT USEFUL rate may be of use as a guide for new integration points. On an abstract level we see negative and positive signals where users have engaged with robot comments, and on that abstract level we can compute a *negative user engagement rate* (negative / negative + positive).

If we consider the signals we have about robot comments in MEAN, we have PLEASE FIX, DONE, ACK, REPLY, QUOTE, and RESOLVED. Among these, we consider NOT USEFUL to be negative (and possibly REPLY), while we consider the rest to be positive, with RESOLVED as the strongest signal. Based on this assessment, we express the negative user engagement rate as NOT USEFUL/ (NOT USEFUL + RESOLVED) (assuming

Analyzer	Published	NOT USEFUL	RESOLVED	User Engagement	% NOT USEFUL	Estimated as Used
Pylint	$11,333$	159	177	3.0%	1.4%	614
Comment-check	$3,185$	196	44	7.5%	6.2%	48
Flake8	$2,273$	0	0	0%	0%	163
Shellcheck	$2,204$	164	29	8.8%	7.4%	44
Macrocheck	$1,117$	40	36	6.8%	3.6%	59
Hadolint	660	12	6	2.7%	1.8%	21
Ansible-lint	62	0	0	0%	0%	8
Total	$20,834$	571	292	2.9%	2.7%	957

Fig. 2. **User Engagement and Use of Results** The number of robot comments published for each analyzer, the number of interactions with these, the user engagement for all comments, % of NOT USEFUL for all comments, and number of results estimated as used.

that REPLY overlaps with NOT USEFUL).

Finding 5. *We found a need for a more general notion of the* NOT USEFUL *rate in the form of a 'negative user engagement rate' (negative / negative + positive).*

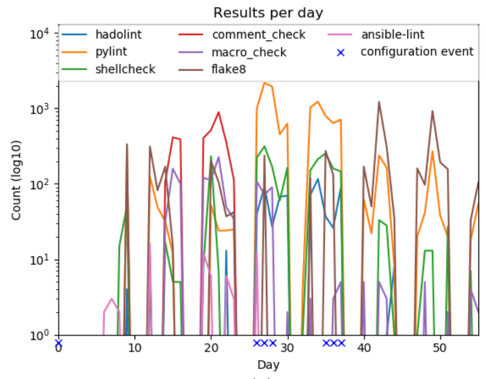
From what we know, the NOT USEFUL rate in Tricorder is computed based on interactions with a small fraction of robot comments, estimated to 0.8% in Section II. We further note that a single NOT USEFUL click for an analyzer will give it a NOT USEFUL rate of 100%. It is unclear how many clicks should be gathered before the rate should be considered stable enough to take into consideration, or if some minimal number of results should have been produced. We see an opportunity for mere guidance here and also exploration of other models, for instance, models based on the number of results produced, or models combining clicks and results. Other examples of models to explore include percentage of not useful results (NOT USEFUL clicks / published results), or percentage of used results (estimated as used / published results).

Finding 6. *There is a lack of guidance for how to use the* NOT USEFUL *rate and it is unclear how it compares to other models for tuning as a signal for system health.*

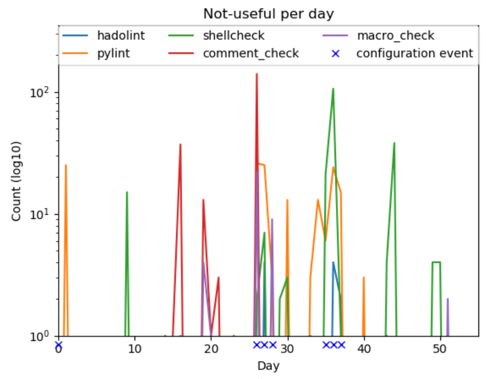
VI. RESULTS: EFFECTS OF DEPLOYMENT

Finally, with our last research question (**RQ$_3$**), we seek to understand the effects of deploying the system presented in Section V.

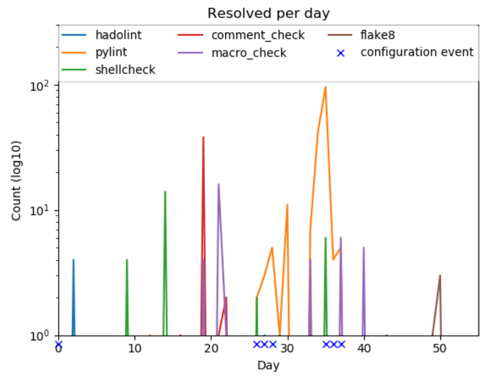
Quantitatively during the deployment, the MEAN system analyzed patchsets from 485 changes, which resulted in $20,834$ analyzer results published as robot comments in Gerrit ($3,497$ on changed lines). A total of 407 users were connected to the analyzed changes, as either change owners or reviewers.



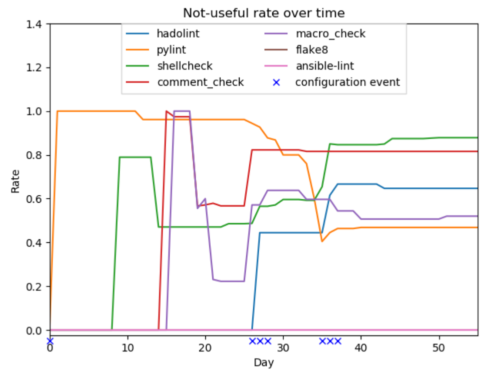
(a)



(b)



(c)



(d)

Fig. 3. Results, clicks in log-10 scale and NOT USEFUL rate over time.

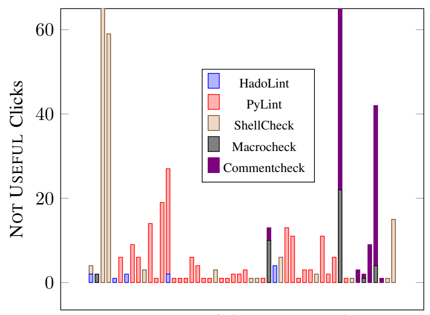


Fig. 4. **Not Useful Clicks per Unique User** per user and analyzer. The two cut off bars have a max of 118 (last is ShellCheck) and 162 (last is Commentcheck). Total number of users is 52.

Figure 2 shows the number of robot comments published for each analyzer and quantitatively how users interacted with these published comments, in terms of Not Useful and Resolved. Figure 3(a) shows the results computed per day and analyzer during the deployment (week 7 coincided with a school holiday).

A. *RQ$_{3.1}$ What is the level of user engagement?*

Considering all published comments and the data gathered for Not Useful and Resolved (listed in Figure 2), we see users engaging with 2.9% of published robot comments (column 5). We note that this is higher than that estimated for the Tricorder system (0.8%). When we consider the percentage of comments receiving Not Useful clicks (Column 6), we see a tendency for increased user engagement (in comparison between analyzers) when the percentage of robot comments with Not Useful clicks go up.

When we inspect the Not Useful and Resolved clicks over time (Figure 3) we see that Not Useful clicks spike at specific times during the deployment and that the higher spikes correlate with times when more results were produced (Figure 3(a)). We further see users engaging with comments via Resolved, especially for PyLint in week 6.

When we look closer at unique users and how they engaged via Not Useful clicks (shown per user in Figure 4), we see that it was provided by 52 users, i.e., 12.8% of users connected to analyzed changes. We see spikes in Not Useful clicks by a small group of users for ShellCheck and Commentcheck. We note that these two analyzers have the highest user engagement, percentage of Not Useful robot comments, and Not Useful rate (Figure 2), and the highest Not Useful click spikes (Figure 3(b)) are also for these two analyzers.

When we consider the total number of interactions with robot comments, for comments anywhere in a change and the subset of comments on changed lines, we see; Not Useful 572 (129), Please Fix 31 (14), Done 223 (128), Ack 39 (8), Reply/Quote 32 (8), and Resolved 292 (145). We note that Not Useful and Reply/Quote were primarily clicked on comments not connected to changed lines (78% for Not Useful and 75% for Reply/Quote).

Finding 7. *We see a user engagement of 2.9% (vs. 0.8% for Tricorder) and Not Useful clicks from 12.8% of users. We further see that Not Useful clicks primarily happened on robot comments outside changed lines, and we see that a small group of users can significantly influence the metrics for an analyzer.*

B. *RQ$_{3.2}$ How many results are used?*

To estimate the number of used results, we used the method outlined in Section III to reduce the total number of published robot comments to 10,640 unique comments. Then we estimated the number of used results per analyzer and change. We identified changes with published robot comments and with more than one patchset, and analyzed each such identified change for each analyzer (identified change * analyzer = 265 cases). We found 121 cases with a decrease (23 had an increase) corresponding to 1,132 potentially fixed robot comments.

To account for configuration changes, we compared the times when patchsets were uploaded with when the configuration changed. We found that a decrease of 175 robot comments was explained by configuration changes. When we remove these comments, 957 robot comments remained among the ones we estimate as fixed, i.e., 9.0% of all unique comments. Figure 2 shows the estimated number of used results per analyzer (Column 7). We note that Flake8 and Ansible-lint have 0% user engagement and at the same time have relatively many used results.

When considering unique comments on changed lines (1,778), we find 513 fixed comments, i.e., er estimate 28.9% of comments on changed lines to be fixed and 5.0% for unchanged lines.

Finding 8. *We estimate that 9.0% of unique results were used and results presented as robot comments on a changed line were more likely to be used (28.9% on changed lines vs. 5.0% on unchanged lines).*

C. *RQ$_{3.3}$ What is the user experience?*

To gather data on the user experience, we sent a survey to developers. Among the 12 respondents, 6 had used results, 2 had clicked Not Useful (and not used results), while 4 had not engaged with robot comments (by either using or clicking). Survey participants were asked whether they agreed with the robot comments on a change; 5 respondents agreed with all robot comments, 5 had a mix of disagreement and agreement, and 1 respondent agreed or had no opinion. No participants only disagreed.

Respondents that had not engaged with robot comments generally agreed with all comments (one respondent in this group had mixed agreement). Among the two respondents that had clicked Not Useful, one agreed with all comments despite having a large number of comments on the selected change (114), and the other had mixed agreement. Among the respondents that had not clicked NOT USEFUL, 3 disagreed with comments and in two cases significantly (32 of 34 comments, 12 of 14 comments).

When asked about why robot comments may be not useful respondents mentioned: false positives (mentioned by 4 respondents), comment on unchanged line (3), low payoff (2), against coding standard (2), irrelevant (1), and repeated comment in the same file (1). Comments on unchanged lines was pointed out by several respondents (6) as not actionable in the context of a Gerrit change. One respondent expressed agreement with all comments in the selected change but ignored them because they were on unchanged lines (*"comments unrelated to the changed code is something that you want to fix in a separate change"*).

We note that we see examples of user behavior with users who completely agree but do not use results, and users who disagree but do not click NOT USEFUL. We see an overlap with the first behavior and comments on unchanged lines. For users not clicking NOT USEFUL we see examples of confusion.

Finding 9. *Comments on unchanged lines stand out as less useful (and sometimes ignored) and at odds with the code review workflow, The* NOT USEFUL *feedback captures usability concerns beyond false positive results.*

D. *RQ$_{3.4}$ What is the maintainer experience?*

The maintainers disabled a total of 20 analyzer categories (PyLint: 9, Shellcheck: 8, Macrocheck: 1, and Commentcheck: 2) via 7 configuration changes (Figure 3), with one very early configuration update and the rest during the middle of the deployment. Figure 3(d) shows the NOT USEFUL rate accumulated over time, where spikes in NOT USEFUL clicks (Figure 3(b)) bring the graphs up, while spikes in RESOLVED (Figure 3(c)) bring the graphs down.

Of the 20 disabled categories, the decisions to disabled 16 were taken due to feedback via NOT USEFUL, guided by the 5% rule (Section III). The remaining 4 categories (all PyLint) were disabled quickly in the early testing of MEAN in the first two weeks of the deployment (*import-error*, *line-to-long* and *[C]onvention* were noticed to have big potential of flooding or false positives), or due to feedback from developers in close proximity to the maintainers. (*wrong-import-order* produced a lot of false positives due to a lack of knowledge of third-party packages in the Docker container wrapping the PyLint analyzer).

We note that the close proximity to the developer infrastructure team, together with the individual assessment of the maintainers, allowed for early tuning of MEAN (first configuration marker in Figure 3(b)). When we consider the deployment to another team (during day 21-41 in Figure 3(b)), we see spikes in feedback for Commentcheck in the beginning and later Shellcheck. We note that these NOT USEFUL spikes correlate with configuration changes (blue markers in the bottom) and we do not see spikes to the same extent afterwards (except for one smaller spike for ShellCheck that may be due to other users giving feedback due to a larger group of users) despite results still being produced for these analyzers (Figure 3(a)).

Finding 10. *The* NOT USEFUL *feedback enabled tuning of MEAN when the distance between users and maintainers increased.*

VII. THREATS TO VALIDITY

Internal Validity - Credibility Robot comments counted as *used*, can also be explained by removed code. There is a risk that the number of used robot comments are estimated higher than the number of robot comments that were fixed, while keeping functionality. Some developers at the company 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. The email-based survey were not anonymous and this may have resulted in participant response bias [11], but we encouraged recipients to provide feedback and we did receive negative feedback.

Analyser selection We picked analysers based on discussions with the dedicated tools team at the company. Different analysers have different response properties, for example with linters being more noisy than other analysers, but more local in their results.

External Validity - Generalisability We designed this study to investigate to what extent Google's philosophy of program analysis is transferable to a different socio-technical context. The results are not fully generalisable to all development contexts, for example, they may not apply to a volunteer-based open source project. However we sought a case study that we think is representative of a broad set of organisations doing development.

VIII. DISCUSSION

The change in the context from Google to different software development organisations has brought with it a different focus. This work places much less emphasis on scale, but more on the need for supporting large variation and the independence of teams (Finding 1). As the social distance between the maintainers of the infrastructure and their users increase the existence of tools to measure and understand the experience of the users becomes crucial (Finding 10). We have seen that NOT USEFUL feedback plays this role by supporting tuning of the system by a centralised group.

The experience the organisation has had with program analysis is similar to that described in the literature [1]–[3]. This has lead to practices where teams reduce the noise by a limited selection of analyzers (Finding 2). However one of the limitations of the approaches typically taken in data-driven analysis so far is that it is easier to measure the costs of program analysis, in false positives and not-useful comments, than to measure the benefits which tend to fall back to rhetorical justifications. Better measures of the benefits of program analysis would be a useful direction for future research.

A further avenue to explore is around measuring engagement. We see a higher user engagement (Finding 7) than we predicted based on the user engagement of Tricorder (Section II). This may be related to the Tricorder estimate being from an average over a year and after longer tuning while ours is from total user engagement after the 8 first weeks of deployment. We tracked additional metrics for estimating used results (Finding 8) and gathered data on the user experience (Finding 9). By splitting data on the level of individual users we see that a small group of users can have a large effect on the metrics (Finding 7).

The focus on the individual user experience with both quantitative and qualitative assessment has highlighted some new usability issues. For example we find comments on unchanged lines to be a significant issue (Finding 7, Finding 9), and comments on changed lines had a significantly higher chance of being used (Finding 8). A key takeaway for how to integrate code review is that comments that the users can action should be prioritized.

Despite being of limited scale, our study gives indications for what the early deployment of the approach may look like. For example, we see stormy periods when new analysers are enabled. The maintainers of the Tricorder system has a practice of putting noisy analyzers on a probation list [4]. Our experience confirms this strategy and we further recommend putting all analyzers on a probation list in early deployment, and the maintainers should be especially attentive to the feedback they receive. We speculate that metrics like the NOT USEFUL rate or the 5% rule may play a bigger role as the system becomes more mature, and the risks become easier to manage, but that during the onboarding process for new analyzers richer feedback mechanisms are needed to manage the existence of unknowns.

Future Work The approach of using mixed methods to understand the experience of developers and to use data about their experience to drive the improvement of the system shows promise. We see several areas for future work:

Firstly: Whilst the precision of the 'used results' metric requires further improvement, we see that the significant

Secondly: The meaning of the NOT USEFUL metric within Tricorder was strongly coupled to its code review integration. We map this to a more general 'negative user engagement' metric (Finding 5), but suggest that the integration within a broader review system may mean that alternative feedback signals are also available. There is a clear gap in knowledge about how to operationalise the feedback signals, especially in early deployment. We explore a different metric, the 5%

majority of results are not being used. Further research is needed to understand this phenomenon.

rule, where we add additional guidance for when to take the metric into account. An application of our mixed methods approach could be used to better understand how to design and operationalise a range of feedback mechanisms at the different stages of development.

IX. CONCLUSIONS

In this paper we have described how we have designed and deployed the MEAN system for data-driven deployment of program analysis on an open tools stack. In doing so, we have replicated some of the success of the Tricorder system in a different context and have also learned more about the user interaction. Our results indicate that the use of NOT USEFUL feedback enables tuning of the system as distance between developers and maintainers grow. We further see a slightly higher user engagement than for the Tricorder system, but also identify integration pain points centered around presenting results on unchanged lines in code review. Our results suggest that the design principles for static analysis, embodied in the Tricorder system and now the MEAN system, may fit best in a context with centralized tooling and with resources available to dedicate maintainers to the system, especially in early deployment.

REFERENCES

[1] B. Johnson, S. Yoonki, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?." *ICSE'13: 35th International Conference on Software Engineering*, pp. 672 – 681, 2013.

[2] 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," *SANER'18: 25th International Conference on Software Analysis, Evolution and Reengineering*, pp. 38–49, 2018.

[3] N. Imtiaz, A. Rahman, E. Farhana, and L. Williams, "Challenges with responding to static analysis tool alerts." *MSR'19: 16th International Conference on Mining Software Repositories*, p. 245, 2019.

[4] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *ICSE'15: 37th International Conference on Software Engineering*. IEEE, 2015, pp. 598–608.

[5] E. Söderberg, "Tricium – Tricorder for Chromium. Early design document." https://bit.ly/tricium-early-design, 2016, accessed on Jan 22, 2021.

[6] Shipshape committers, "Shipshape," https://github.com/google/shipshape, accessed on Jan 22, 2021.

[7] M. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran, "Making defect-finding tools work for you." *ICSE'10: 32nd International Conference on Software Engineering*, vol. 2, pp. 99 – 108, 2010.

[8] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study reseach in software engineering," *Empirical Software Engineering*, vol. 14, 2008.

[9] H. Munir, P. Runeson, and K. Wnuk, "Open tools for software engineering: Validation of a theory of openness in the automotive industry," in *EASE'19: Evaluation and Assessment on Software Engineering*. ACM, 2019, pp. 2—11.

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

[11] N. Dell, V. Vaidyanathan, I. Medhi, E. Cutrell, and W. Thies, ""Yours is Better!": Participant response bias in HCI," *CHI'12: Conference on Human Factors in Computing Systems*, p. 1321–1330, 2012.