



Integrating Mutation Testing Into Developer Workflow: An Industrial Case Study

Downloaded from: <https://research.chalmers.se>, 2024-12-19 18:20 UTC

Citation for the original published paper (version of record):

Van Heijningen, S., Wiik, T., Gomes, F. et al (2024). Integrating Mutation Testing Into Developer Workflow: An Industrial Case Study. 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)

N.B. When citing this work, cite the original published paper.

© 2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, or reuse of any copyrighted component of this work in other works.

Integrating Mutation Testing Into Developer Workflow: An Industrial Case Study

Stefan Alexander van
Heijningen
Theo Wiik
Chalmers and University of
Gothenburg
Gothenburg, Sweden
stefanva@student.chalmers.se
withéo@student.chalmers.se

Francisco Gomes de Oliveira
Neto
Gregory Gay
Chalmers and University of
Gothenburg
Gothenburg, Sweden
francisco.gomes@cse.gu.se
greg@greggay.com

Kim Viggedal
David Friberg
Zenseact
Gothenburg, Sweden
kim.viggedal@zenseact.com
david.friberg@zenseact.com

ABSTRACT

Mutation testing is a potentially effective method to assess test suite adequacy. Researchers have made mutation testing more computationally efficient, and new frameworks are regularly emerging. However, there is still limited adoption of mutation testing in industry. We hypothesize that such adoption is hindered by a lack of guidance on how to effectively and efficiently utilize mutation testing in a development workflow. To that end, we have conducted an industrial case study exploring the *technical* challenges of implementing mutation testing in continuous integration, *what* information from mutation testing is of use to developers, and *how* that information should be presented (in textual and visual form). Our results reveal five technical challenges of integrating mutation testing and nine key findings regarding how the results of mutation testing are used and presented. We also offer a dashboard to visualize mutation testing results, as well as 16 recommendations for making effective use of mutation testing in practice¹.

CCS CONCEPTS

• **Software and its engineering** → **Software verification and validation**; *Collaboration in software development*; *Software configuration management and version control systems*.

KEYWORDS

Mutation Testing, Test Adequacy, Software Visualization, Software Testing

ACM Reference Format:

Stefan Alexander van Heijningen, Theo Wiik, Francisco Gomes de Oliveira Neto, Gregory Gay, Kim Viggedal, and David Friberg. 2024. Integrating Mutation Testing Into Developer Workflow: An Industrial Case Study. In

¹Support provided by Software Center project “Trustworthy and Human-Centered Test Automation” and a Lars Pareto travel grant from Chalmers and University of Gothenburg. We also thank all participants from Zenseact.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASE ’24, October 27–November 1, 2024, Sacramento, CA, USA

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-1248-7/24/10

<https://doi.org/10.1145/3691620.3695273>

39th IEEE/ACM International Conference on Automated Software Engineering (ASE ’24), October 27–November 1, 2024, Sacramento, CA, USA. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3691620.3695273>

1 INTRODUCTION

Ensuring that software works as intended is crucial, especially in safety-critical systems where faults can lead to severe consequences [31]. Testing is a common method of assessing the behavior of software, based on the application of selected input and inspection of the resulting behavior [2]. A natural question for developers to ask, however, is when they have conducted “enough” testing. To help answer this question, developers measure the strength of a test suite using adequacy criteria, i.e., measurements of how thoroughly the codebase-under-test has been exercised by its test suites [6].

Mutation testing is a practice where artificial faults (“mutants”) are seeded into the codebase [29]. Test adequacy can then be assessed by measuring how many mutants were detected by the test suite. Mutation testing is potentially one of the most robust adequacy criteria [26]. However, it has not yet been widely adopted in industry due to (i) its computational cost when re-executing tests for each mutant, and (ii) the immaturity of mutation testing frameworks, especially for languages other than Java [25, 27].

Recently, significant effort has been made to reduce the computational cost of mutation testing [8, 18, 23, 38]. At the same time, availability of open-source mutation testing tools has increased, suggesting that mutation testing is reaching a sufficiently mature state to be applied in practice [16, 26]. This is also evidenced in recent research conducted in industrial settings [4, 25, 28, 35]. Computational cost and tool immaturity are still significant hurdles but may no longer be the primary barriers preventing adoption.

Rather, we hypothesize that a lack of best practices and guidance regarding how to integrate mutation testing into development workflows and how to make use of its results to improve test quality hinders its adoption.

This challenge has three dimensions that must be addressed. First, on a *technical* level, few have explored the challenges of implementing mutation testing within the automated build systems and continuous integration (CI) pipelines [25, 27]. Moreover, even if mutation testing can be executed, there is an education gap [4, 28, 35]—how should developers actually interpret and apply the results of mutation testing? Therefore, two additional dimensions

that must be addressed include *what* information from mutation testing is of use to developers and *how* that information should be presented (e.g., in textual or visual form) to maximize its relevance, comprehensibility, and applicability.

We explore these three dimensions in a case study at Zenseact, a company developing Autonomous Driving software. We have implemented the Mull mutation testing framework [9] into a nightly CI pipeline at Zenseact, and have used observations on this process to produce an experience report discussing the technical challenges encountered and their solutions. We also developed a dashboard where we visualize the results of mutation testing at different levels of granularity such as the team, directory, and file level. We then conducted a series of think-aloud observations and semi-structured interviews with Zenseact developers where they used both the dashboard and a text-based report on the mutation testing results to investigate test suite quality.

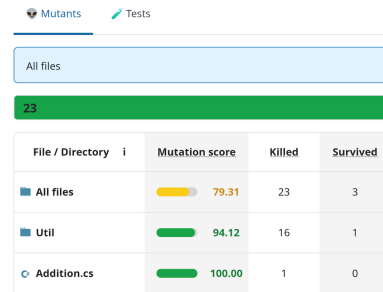
Our case study illustrates the technical and human-based challenges that emerge when applying mutation testing in an industrial setting. Particularly, we identified five technical challenges, including tool immaturity, issues emerging from the combination of mutation testing framework, codebase, and build system, and the integration process itself. We also offer recommendations for addressing each challenge. Based on the observations and interviews, we also present nine findings and sixteen subsequent recommendations regarding mutation testing information and result presentation including the importance of gaining an overview of test suite quality and its evolution, the information useful to different stakeholders, missing contextual information, the value of filtering mutation testing results for different levels of granularity, effective visualization, and the continuing need for education on interpreting mutation testing.

2 BACKGROUND

Mutation Testing is a technique used to assess the sensitivity of a test suite to small changes in the code [29]. To do so, automated code transformations are applied to produce faulty versions of the codebase-under-test (**mutants**). Generally, each mutant contains a single modification, imposed by a **mutation operator**. Each mutation operator reflects a repeatable program change, such as changing an expression (e.g., substituting addition for subtraction), that can be automatically applied to statements that fit the correct pattern. Mutation operators are modeled after simple faults that could appear in a program [12].

The effectiveness of a test suite can be assessed by examining how many are **killed** (that is, detected) by the suite. The **mutation score** is the ratio of killed mutants to the total number of mutants. The mutation score can be considered an indicator of the strength of the test suite, and certain thresholds may be targeted [29].

An issue traditionally hindering adoption of mutation testing is its prohibitive cost, as the test suite would need to be re-executed on each mutant. Prior work has identified three viable strategies for applying mutation testing [18, 25]. First, it could be executed *manually* when needed. Second, it could be applied only to code *changed in a commit*. Third, it could be applied *periodically* when computational resources are available, e.g., nightly. In this study, we focus on periodic application.



The screenshot shows a dashboard with two tabs: 'Mutants' and 'Tests'. The 'Mutants' tab is active. Below the tabs, there's a summary bar for 'All files' showing a total of 23 mutants. Below this is a table with columns: 'File / Directory', 'Mutation score', 'Killed', and 'Survived'.

File / Directory	Mutation score	Killed	Survived
All files	79.31	23	3
Util	94.12	16	1
Addition.cs	100.00	1	0

Figure 1: Excerpt of a report generated by Stryker Mutator.

Mutation testing tools typically present their results by generating **reports**. The Stryker Mutator project offers an open-source schema (Mutation Testing Elements [20]) for such reports, as shown in Figure 1. The report contains information such as the mutation score at different levels of granularity (project, directory, and file), what mutation operators were applied to specific lines of code, and whether each mutant was killed or survived.

Data Visualization is a technique for displaying information graphically to facilitate its interpretation or analysis [34]. Visualization is common in software development and can convey information related to, e.g., code coverage [24], software performance [15], and requirement traceability [17]. Data visualizations are often displayed to users within a **dashboard**, i.e., an interactive format aggregating data and visualizations from different sources [33]. In this study, we make use of the Kibana dashboard [21] developed by Elastic. Kibana dashboards are commonly used to visualize logs recorded by software infrastructure monitoring tools.

3 RELATED WORK

Much of the research on mutation testing has focused on its effectiveness (e.g., [12, 13]) or computational cost (e.g., [19, 22]). Our primary focus is on how mutation testing should be integrated into development workflow. We previously conducted a case study at Zenseact where we assessed the feasibility of existing C++ mutation testing frameworks for integration into a CI pipeline [25]. We also conducted an interview study to explore how developers would apply mutation testing, resulting in a set of recommendations, including a need for education, to visualize trends over time, and to offer flexible trade-offs between scalability and level-of-detail of the results. In this study, we build on and apply a subset of our recommendations in practice.

Others have also examined aspects of integrating mutation testing in industry. For example, Parsai et al. explored the viability of integrating mutation testing into build systems, noting multiple challenges, but ultimately finding that the computational overhead could be managed with effective tool use [27]. However, few have explored how to best present mutation testing results to developers. Authors have observed that the number of mutants in a large-scale project makes it infeasible—in both cost and mental burden—to inspect each individual mutant [4, 28, 35]. Beller et al. also note a lack of education on how to use mutation testing effectively [4]. Vercammen et al. conducted an industrial case study with two companies—one with five years of mutation testing experience and one without experience [35]. The inexperienced company struggled to use mutation testing effectively and felt that the cost of tool

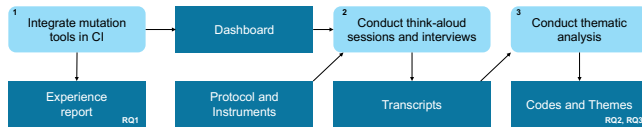


Figure 2: Overview of our research method.

maintenance was high. The experienced company felt that mutation testing was effective, but only if its results can be provided in a simple, interpretable, and scalable form.

Two common solutions, primarily focused on the computational cost, are to reduce the number of employed mutation operators [28] or to focus on commit-relevant mutations [18, 23]. At Google, in addition to operator and commit-level limitation, a set of heuristics are used to hide “unproductive mutants” [28]. Parsai et al. also visualized mutation scores aggregated per class [27]. More work is needed in this area.

Adler et al. explored how to present code coverage results to developers [1]. They use a data presentation technique called “substring hole analysis”, which clusters coverage information based on code elements with similar names. Thus, syntactically-linked data can be selectively presented to the user, while omitting unrelated coverage data. They find that this technique made code coverage analysis more cost-effective on industrial code bases. Collectively, these results suggest that research is still needed on how to integrate mutation testing into development workflows and on how to present and interpret its results.

4 RESEARCH METHODOLOGY

In this study, we address the following research questions:

RQ1: What challenges arise when integrating mutation testing tools, and how can they be addressed?

RQ2: What information from mutation testing should be presented to developers?

RQ3: How should information from mutation testing be presented to developers?

RQ1 focused on the challenges that hinder the *technical integration* of mutation testing such as dependency conflicts, inability to interface with existing tools, and maintenance efforts. **RQ2–3** focus on *developer effectiveness*, exploring what information is most useful (e.g., mutation score, mutation operator breakdown) and how this information should be conveyed (e.g., in textual or visual formats). To answer those questions we conducted an exploratory and interpretivist case study (Figure 2) following the terminology defined by Runeson et al. [32] and Baltes et al. [3].

We began by integrating a mutation testing framework, Mull [9], into a CI pipeline at Zenseact. In parallel, we developed a dashboard to visualize the data in the report generated after executing the framework. We report on our observations regarding the issues encountered and how they were handled in Section 4.2, hence producing recommendations to address RQ1. Next, we collected qualitative data from developers in a session composed of two parts. First, we conducted think-aloud sessions with developers, where participants were tasked with assessing test suite quality

Table 1: Protocol to register observations for RQ1.

Field	Description
Timestamp	Date and time of the event.
Names	Names of people conducting and participating in the observation.
Event	One of the following: Decision, Issue Encountered, Issue Solved, Doubt Raised, Progress Made, Sentiment, Reflection, Discovery, Meeting
Observation	Detailed information on the event.
Next Steps	Activities that should follow the event.

using the HTML mutation report and the created dashboard. We collected observations on developer’s interactions and opinions on the dashboard and report. The second part of this session was semi-structured interviews with the developers who participated in the think-aloud sessions, focusing on the perceived benefits and issues. We recorded the entire sessions and then used thematic analysis to analyze the transcripts, our observations, and the interview data to answer RQ2–3 (Section 4.3).

4.1 Case Study Context

The case study was performed at Zenseact, a Swedish company with approximately 600 employees [37]. Zenseact develops Advanced Driver Assistance Systems and Autonomous Driving software, primarily written in C++. Because automotive software is safety-critical and must conform to various standards (e.g., ISO 26262 [7]), verification and validation are important. As such, they conduct extensive testing, including manual review, exploratory testing, and automated testing in CI pipelines.

Mutation testing is not currently a standard practice at the company, but many developers were somewhat familiar with the concept [25]. The participating developers did not have a specific goal for the integration of mutation testing, but were broadly interested in how it could be used to assess and improve test suite quality.

Zenseact utilizes the concept of *guardianship*, where a team is the main owner of a part of the code [10]. During this study, guardianship information was incorporated into the developed dashboard to help contextualize mutation testing results.

4.2 Mutation Testing Integration (RQ1)

In a previous study, we identified Dextool [5] and Mull [9] as viable C++ mutation tools [25] for CI. Following initial experimentation, various issues arose when using Dextool in Zenseact’s CI pipeline, such as dependency clashes between the mutation framework and codebase. Therefore, we focused on integrating Mull into a CI pipeline for nightly builds of Zenseact’s C++ codebase.

We followed the guidelines by Hancock et al. [11] to define a protocol to systematically collect observations during the integration. To form the experience report, those observations were sequentially analyzed and discussed with Zenseact developers for confirmation, validation, and reflection. The protocol (Table 1) and the input from developers allowed us to gather data and reflect on our integration process to yield the insights reported in Section 6.1.

Table 2: Overview of study participants, including their experience at Zenseact and overall (Over.) in years (y) or months (m), how often they engaged with testing activities, and experience with using Kibana.

ID	Role	Experience		Test Freq.	Kibana
		Zenseact	Over.		
P1	AI support tool developer	1.5 y	3-4 y	Rarely	Minimal
P2	Computer vision engineer	7 y	7 y	Daily	None
P3	C++ toolchain maintainer	3.5 y	12 y	Daily	Skilled
P4	C++, Java developer	7 y	8 y	Semi-daily	Skilled
P5	C++ developer and system architect	2 y	10 y	Semi-daily	None
P6	C++ feature developer	2.5 y	7 y	Daily	Skilled
P7	C++, Python developer	1 m	10 y	Weekly	None
P8	C++ feature developer	1 y	6 y	Daily	Minimal
P9	C++ feature developer	2.5 y	5 y	Weekly	Minimal
P10	C++ feature developer	2.5 m	10 m	Rarely	Minimal

4.3 Mutation Testing Information and Result Presentation (RQ2–3)

After implementing our mutation testing dashboard, we conducted think-aloud observations and semi-structured interviews with developers from Zenseact to determine (i) what information developers perceived as useful from mutation testing, and (ii) how that information should be best presented to developers. Before starting each session, we explained the purpose of the study and asked participants for consent to record their audio and screen interactions. Participants were given the option to opt out of the study at any time, and before publication, a draft of this study was sent to all participants so that they could provide feedback and corrections.

Sampling and protocol: Because our sampling frame is Zenseact, our population of interest consists of software developers in the automotive industry. We aimed for a diversity of roles, time at the company, experience, and degree of interaction with the testing process. The list of participants is shown in Table 2. The think-aloud and interview protocols were developed by the first two authors and reviewed by the third and fourth authors. To ensure clarity and effectiveness, we conducted three pilot studies: two with the Zenseact-based authors of this study and one with a Zenseact developer unfamiliar with mutation testing. This helped verify that the explanations and instructions were clear and understandable.

Session format: Each session lasted approximately 1.5 hours and began with an introduction to mutation testing, followed by a Q&A session for any clarifications. Participants first examined a text-based mutation testing report and a dashboard we provided. They were then tasked with completing a set of activities while verbalizing their thoughts. After a brief break, we conducted interviews where participants reflected on mutation testing and the different presentation formats.

Think-aloud procedure: Before the tasks, participants received a guided tour of the dashboard and report to familiarize themselves with the tools without influencing their inspection. Each participant was then given a list of tasks, detailed in Table 3, to assess test quality using the report and dashboard. Tasks focused on a specific team, directory, and file, but participants could later explore mutation testing results for the entire codebase, including their own team. To emphasize core mutation testing concepts, we limited the scope

Table 3: Think-aloud tasks and interview instrument. *Alpha*, *Beta*, and *Gamma* represent a team, directory, and file, renamed for confidentiality. The same team, directory, and file were used for all participants.

Demographic Questions	
1	What is your role at Zenseact?
2	How long have you worked at Zenseact?
3	How many years of experience do you have as a developer?
4	How often do you interact with unit tests?
5	What is your impression of mutation testing, after the presentation?
6	Do you have any experience using Kibana dashboards?
Think-Aloud Tasks	
1	Familiarize yourself with the dashboard and report.
2	Assess the quality of tests for team <i>Alpha</i> .
	(a) In which direction is test quality evolving for the team?
	(b) Is test quality consistent across the directories the team maintains?
	(c) Are there any directories that require more extensive testing?
3	Assess the quality of tests for directory <i>Beta</i> .
	(a) In which direction is test quality evolving for the directory?
	(b) Is test quality consistent across the files in the directory?
	(c) Are there any files that require more extensive testing?
4	Assess the quality of tests for file <i>Gamma</i> .
	(a) Are surviving mutants concentrated around the same code regions or spread out?
	(b) If you could make or modify a test for this file, what would you address?
	(c) In which direction is test quality for the file evolving?
Interview Instrument	
Questions related to information (RQ2)	
1	Did you find any particular information useful?
2	Did you find any particular information not useful?
3	Was there any particular information missing?
Questions related to visualizations (RQ3)	
1	Do you think there is a better way to display any information presented?
2	Was there any information or visualization you found confusing or hard to interpret?
3	Did you find any particular information missing?
4	How did you find the quantity of information provided in the report and dashboard?
5	How difficult was it to find the information needed, given your tasks?
6	Do you think the experience of using the report and dashboard would change if significantly more teams, directories, and files were involved? If so, how?
7	How was the experience of using both the dashboard and the report?
Concluding questions (RQ2–3)	
1	Do you believe the potential benefits of mutation testing are worth the time and effort required to learn to use the dashboard and report? Why or why not?
2	How do you believe your experience would have changed if you had to complete the tasks using only the dashboard or the report?
3	Would you use the report and dashboard if introduced at Zenseact? Why/why not? What would you mainly use? Why?
4	Do you have any final thoughts you wish to share?

to a subset of mutation operators: arithmetic operator replacement (e.g., $+ \rightarrow -$), shortcut operator replacement (e.g., $x++ \rightarrow x--$), and relational operator replacement (e.g., $< \rightarrow \leq$). Throughout the tasks, participants were encouraged to articulate their impressions, interpretations, and opinions on the visual aspects. If they were silent for extended periods, we prompted them to describe their actions and thoughts. Clarifications on mutation testing, the report, or the dashboard were provided as needed.

Interview study: The interview questions, shown in Table 3, were organized into three sections: information, visualizations, and final reflections. The questions aimed to gauge the perceived usefulness and interpretability of mutation testing results. As the

interviews were semi-structured, we also posed follow-up questions to elicit further insights.

Reflexive thematic analysis: Reflexive thematic analysis is an inductive process, where codes were created and refined to analyze qualitative data [36]. To aid the analysis, we used a journal to reflect on the coding process and examine our assumptions. The noted assumptions were: “people do not often want to learn new tools”, “participants might appear positive towards the developed solution as they want to be nice”, “participants understand code”, and “participants understood mutation testing”.

Voice recordings were transcribed using an AI-based speech recognition tool [30], then manually corrected. Screen recordings were used to clarify voice recordings. During the coding process, we highlighted relevant parts of the transcripts and assigned code labels—short identifiers—to each. We then developed codes describing each highlighted segment. We had multiple discussions and iterations of the codes and code labels grouping them into themes and sub-themes. This process was completed by the first two authors, with feedback from the other authors.

We used Krippendorff’s α to assess inter-rater reliability [14]. The first two authors independently coded one session to identify overlapping or contrasting codes between both authors, hence discussing both agreements and disagreements in the coding process. Due to a low agreement rate ($\alpha = 0.45$, with a 95% confidence interval of $[0.25, 0.61]$), authors independently coded a second session which, then, showed sufficient agreement between authors ($\alpha = 0.72$, with a 95% confidence interval of $[0.54, 0.86]$).

5 TECHNICAL DELIVERABLES: CI INTEGRATION AND DASHBOARD

In this section, we present two technical deliverables from our research. The first is the integration of Mull, a mutation testing tool, into a CI pipeline for nightly builds, and the second is the dashboard designed to display and interpret mutation testing results.

5.1 Integrating Mull and Nightly Builds

The operations executed in the CI pipeline and their resulting artefacts are shown in Figure 3. Our integration included code and scripting (i) to execute Mull, (ii) to handle issues such as the build system failing, and (iii) to format and upload the results of executing mutation testing for developer use.

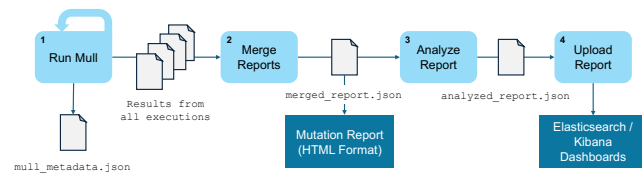


Figure 3: Operations in the CI pipeline and by-products.

Each time mutation testing is performed, our scripts collect: (i) timestamp from the build, (ii) the commit in which the build was executed on, and (iii) mutation score. For each mutant, we extract whether the mutant survived or was killed, the line of code mutated, and which mutation operator was used. Two formats were used

Team Dashboard

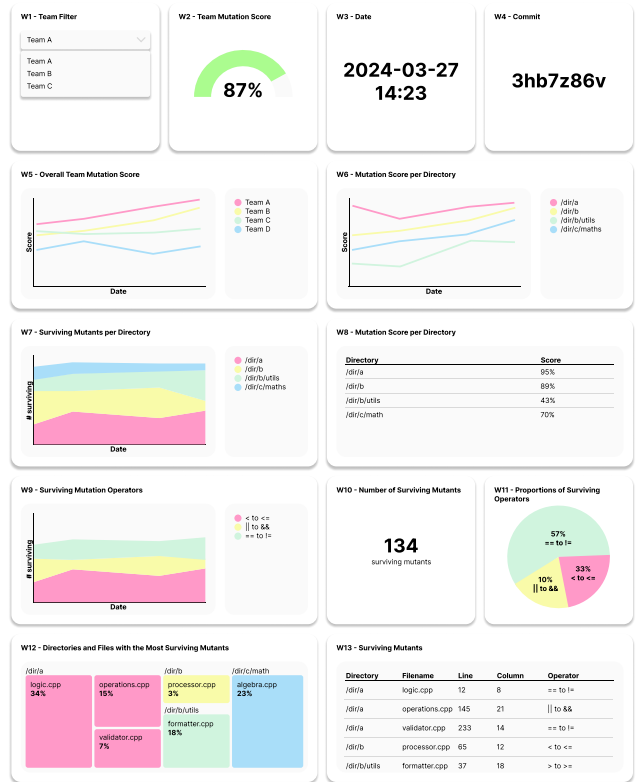


Figure 4: Example team-level dashboard with synthetic data.

to present this information to developers: a text-based report and a dashboard. The text-based report is directly generated by Mull, conforming to the Mutation Testing Elements schema [20]. Mull produces one report per test suite². Each time we execute mutation testing, we execute it for all suites, then merged the generated reports into a single aggregate report.

5.2 Designing the Mutation Testing Dashboard

We also developed a Kibana dashboard [21] presenting current mutation testing results, and the how mutation testing results have changed over time. We chose Kibana because it is already used at Zenseact, hence practitioners are already familiar with using such dashboards as part of development. Note that this choice also limits the supported visualization types.

The dashboard—an anonymized illustration is shown in Figure 4—consists of a set of interactive **widgets**—specific visualizations of underlying information (e.g., mutation score or lines of code in a file) detailed in Table 4. The dashboard allows users to filter results to a particular team, directory, file, or mutation operator. As previously explained, Zenseact utilizes guardianship. A user could select their team, then they would be presented with a dashboard containing results for the directories “guarded” by their team.

²We use the term “test suite” to refer to a C++ file containing one or more test cases.

Table 4: Widgets implemented in the dashboard.

Name	Visualization	Description
Filter	Dropdown	Filter values by team, directory, or file.
Current Mutation Score	Gauge	The current mutation score for the chosen filter.
Date	Single Value	Timestamp of current results.
Commit	Single Value	The commit the current analysis was run on.
Overall Mutation Score	Line Chart	Overall mutation score over time for the chosen filter.
Number of Surviving Mutants	Single Value	Number of surviving mutants at the chosen level of filtering.
Surviving Mutants Per Directory or File	Stacked Area Chart	Number of surviving mutants per directory or file, depending on the chosen level of filtering.
Mutation Score per Directory	Line Chart	Mutation score over time for all directories.
Mutation Score per File	Table	Mutation score per file within a chosen directory.
Surviving Mutation Operators	Stacked Area Chart	Number of surviving mutants per operator over time, for the chosen filter.
Proportions of Surviving Mutants Per Operator	Pie Chart	The proportion of surviving mutants in each mutation operator, for the chosen filter.
Directories and Files with the Most Surviving Mutants	Tree Map	Hierarchical view of files with the most surviving mutants, for the chosen filter.
Surviving Mutants	Table	The surviving mutants, including their location and mutation operator, for the chosen filter.

6 RESULTS

6.1 RQ1: Mutation Testing Integration

Here, we summarise the main challenges and recommendations found when integrating Mull into a CI pipeline for nightly builds. One observation we made was that most of the knowledge required to execute Mull on the codebase was related to the codebase and its build system, rather than the tool itself. Expertise in mutation testing was not required for this integration—rather, it was more important to be familiar with the build system.

Executing Mull on the Full Codebase: Our next step was to apply Mull to all test suites, rather than to a single test suite. To detect and solve issues early, we followed an incremental strategy by starting with a small subset of mutation operators and test suites and expanding gradually. This strategy led to the discovery that specific code operations could cause Mull to crash, including the use of hardware-specific and CUDA code, which is used to execute operations on GPUs. To account for errors, we implemented scripting that would scan the codebase for test suites, then execute Mull on each in sequence, moving on to the next test suite and logging output for inspection if Mull crashed.

Mull must be executed for each test suite, which means that a report is generated for each suite as well. Our goal was to generate a single report for the entire codebase, which meant that we needed to merge multiple reports into a single one. This was an unexpected limitation of the usability of Mull—most industrial-scale codebases will contain multiple test suites.

When a test suite is executed by Mull, all code invoked, directly or indirectly³, by that test suite is mutated. This means that the same mutants⁴ could be used in the evaluation of multiple test suites. To reduce computational costs and reduce ambiguity in assessing mutation score when merging reports, we chose to only assess mutation score on code directly invoked by each test suite. Mull has the ability to specify a whitelist of files to mutate. We generate a new whitelist for each test suite. This decreased the execution time for mutation testing on the codebase by approximately 40%.

Presentation of Results: The HTML reports generated by Mull were not in a format that could be directly uploaded into the dashboard. We also needed to include information not already in the

report, such as code guardianship. We developed scripting to convert the report into a suitable format.

One observation made at this stage was that the dashboard is independent of the specific mutation testing framework. Given a wrapper to convert its output into the correct format, the mutation testing framework used in the backend becomes interchangeable. This adds flexibility to the integration.

Integration Into CI: We performed the initial development on a local machine, then integrated Mull and the scripting into the CI pipeline afterwards. This led to the discovery of behavioral differences between the two environments, related to how the build system (Bazel) and Mull execute.

Mull performs mutation testing in two separate steps: compilation and execution. In the compilation step, Mull records the location of each mutant generated, including the file path, line, and column. During the second step, Mull executes tests and checks whether each mutant survives. To build a target, Bazel moves all files into a sandbox. When building locally, the file system inside the sandbox remains unchanged between steps. However, in CI, targets were compiled inside the sandbox and then executed outside. Mull saved references to file paths inside the sandbox that were no longer valid. To solve this issue, both compilation and execution of test targets had to be performed inside the sandbox.

Retrospective: Most of the time dedicated to integrating Mull into the CI pipeline was related to developing scripting to circumvent unexpected issues caused by the unique combination of codebase, build system, and mutation testing framework. Although Mull was “usable” from the first day, a number of issues had to be addressed to actually integrate the tool. We hypothesize that more work was required to integrate Mull than would be required to integrate code coverage tools into the same pipeline.

Although Mull is suitable for CI integration, we do not consider the tool mature enough to be easy to integrate into a complex codebase. The previous issues encountered with Dextool also suggest the maintenance issues that could occur over time. Mutual dependencies of the codebase, mutation testing framework, and build system, such as Clang or LLVM, must remain in relative lockstep. The effort required to integrate and maintain mutation testing is not insignificant. Therefore, the value provided by mutation testing must be more than the cost of its integration and maintenance for the technique to gain traction in an industrial environment.

³E.g., calls to dependencies of the code directly tested by a test suite.

⁴Mutants in the same file, line, and column, mutated with the same mutation operator and replacement value.

Table 5: Description of themes (bolded) and sub-themes.

Theme	Description
Capabilities	What provided information enables users to do
Summarize	Ability to see the overall status of the test suite
Improve Test Suites	Ability to improve test suite quality
Indicate Source Code Quality	Ability to estimate source code quality
End Users	Who the solution could provide value to
Teams and Team Managers	Value of mutation testing for teams and managers
Developers	Value of mutation testing for developers
Complementary Components	The relation between the report and dashboard
Valued Information	What information was perceived as useful or missing
Score and Surviving Mutants	Perception of mutation score and surviving mutants
Source Code Context	Desire for more context regarding source code
Surviving Mutant Context	Desire for more context regarding surviving mutants
Visual Elements	Which factors were important when visualizing information
Scalability	How developers perceived solution scalability
Simplicity	Developer preferences regarding visualization complexity
Evolution and Trends	Thoughts on showing mutation testing results over time
User Experience	Which factors were important when using the solution
Filtering	Ability to filter information in the dashboard
Navigability	Interconnectedness of elements of the solution
Learning Curve	The learning curve associated with both the overall solution and mutation testing in general
Usability	Ease of use of the solution
Cognitive Workload	Mental effort required to use the solution

6.2 RQ2 and RQ3: Visualising Mutation Testing

An overview of the themes and subthemes extracted from the thematic analysis is presented in Table 5. Note that *the “solution”* refers to the combination of the HTML report and dashboard.

Capabilities: Many of our participants shared how the information provided by our solutions empowers users. The dashboard was particularly useful for *summarising* the state of testing for the entire codebase, i.e., getting a general impression of how test quality was evolving, and enabling them to locate code regions that needed improved testing the most. Moreover, eight participants pointed out that the details from the report such as the surviving mutants, helped them to *improve the test suite*.

“I think the use case for the report is that you will get very detailed information on which mutants were not killed... I would like to improve the tests so that the mutations get killed, if possible.” - P7

Another capability brought up by experienced developers was that mutation testing gave an *indication of source code quality*, in addition to test quality. They speculated that code regions dense with mutants could be taken as code smells or indicative of technical debt. They argued that such complex regions could be refactored to improve readability or testability.

“I think this framework is also quite good to give hints that you have some poor logic. Especially, for instance, if you do a lot of comparisons in one statement, then it’s very error-prone. It could help with refactoring the source code.” - P3

End Users: Our analysis also revealed that both *teams and managers* can benefit from the developed solution. Nine participants

thought the dashboard, in particular, could offer observability into the state of the testing process for non-technical stakeholders such as product owners (PO) or team managers, as no in-depth knowledge of the code was required to interpret the dashboard.

“To give an overview for people that do not work with the specific functions, maybe the dashboard is better.” - P7

Many participants considered the dashboard and report to be *complementary components* of their toolkit. Participants shared that *developers* would use the report more often than the dashboard. Their main focus is on the code and the tests they are responsible for, and need more than just the overview. Participants also felt that knowledge about the code was required to use the report effectively.

“I think I would be able to do everything with only the report. But it would be much more difficult for me to navigate because the visualization [in the dashboard] makes it easier to pinpoint where you want to look.” - P1

Valued Information Participants shared some examples of information that they felt was useful or was missing from the dashboard. The majority of participants indicated that they mainly used the *mutation score and surviving mutants* to assess test suite quality. Less focus was given to the number of surviving mutants, the distribution of mutation operators applied, or other metadata.

When asked what additional information could be beneficial for improving test suite quality, nine out of ten participants discussed information related to the code-under-test. Examples of desired *source code context* included relating mutation testing results to code coverage, file sizes (in lines of code), code criticality, code complexity, and compliance to specific safety standards. Participants also desired *surviving mutant context*, mainly to prioritize which to address first.

“If you could somehow include the [coverage] report, then you could have all testing in one place, which would be nice. At least double check that there is a test that covers [the same code] or not ... Then you could quickly know if the mutant survives because I didn’t have a test or my test was bad. Maybe that could help me investigate more quickly what I should do.” - P6

“If you want to improve your code you would want the tool to show the [mutants] that you should focus on first ... You want to remove the most dangerous first and, maybe, eventually go down to zero.” - P5

Visual Elements: This theme encompasses the relevant factors when visualizing information. Eight participants noted that the dashboard’s visualizations fulfilled their purpose despite the large scale of the codebase. They mainly attribute *scalability* to the filtering functionality, which allows them to see only relevant information in the visualizations.

“Since it’s possible to go to particular directories, I don’t think [scalability] is a big problem, at least in our code base.” - P3

Participants preferred *simplicity* in the visualizations over more detailed, but complex, ones. For example, there is significant hierarchy in mutation testing results—teams guard directories, which contain main files, each with surviving mutants. Participants preferred a separate visualization for each level in the hierarchy instead of one that shows information related to multiple levels at once (e.g., tree maps). Participants frequently praised visualizations that *evolution and trends* of the mutation testing results. This helped

them see how test quality changed over the development process and the impact of their changes.

“[The tree map] is a bit hard to read. Maybe one can do something else there.” - P7

“[The line chart] is quite good, the surviving mutants per directory over time. You’re seeing how the directory is getting affected.” - P1

User Experience: This theme relates the the user experience of the solution. The possibility of *filtering* the data—available in the dashboard, but not the report—was considered very important to avoid overwhelming information.

“I think since we have quite a huge code base, I don’t know how useful these are without filtering.” - P3

Participants frequently switched between the dashboard and the report, highlighting the importance of *navigability*. A common point brought up was that they lacked a way to easily switch between the two — they would find a point of interest using the dashboard and manually navigate to the report. They requested, e.g., a button to directly move from the dashboard to the same point in the report. Some also argued in favor of combining the dashboard and report into a single component.

“If I could press on the link and open [the report] to see it directly ... some kind of link from this page to this” - P2

Seven participants reflected that, even with the initial presentation, they experienced a steep *learning curve* to understand mutation testing or use the solution. They did not find the mutation score intuitive to interpret, unlike a code coverage score. Many wanted to see the results for other teams to get a notion of what a “good” mutation score would be. Regarding the solution, some interactive elements were not straightforward to interpret.

“I feel like it is a bit hard to know what is a good value. I guess there would be guidelines ... like code coverage.” - P10

Participants reflected positively on the *usability* of our solution, explaining that it was easy to use after some guidance and familiarization. The participants did not consider the amount of information displayed overwhelming. Again, the ability to filter data was seen to assist, reducing the mental fatigue associated with analyzing mutation testing results.

“It was great getting some hands-on experience and it was easier than I expected ... When you first hear about [mutation testing], it seems like a complex concept, but it’s really quite simple actually ... with the help of the tools, I mean. Otherwise, if you just get some kind of console output, then that would be much harder.” - P6

“Coming in, I expected to be overwhelmed, but it was not overwhelming and it felt like a good amount [of information], actually. Usually, it takes a while for you to learn how to use these [tools] efficiently and what to look for. But, it felt pretty good initially.” - P5

7 DISCUSSION

7.1 Mutation Testing Integration (RQ1)

We identified challenges and offer recommendations regarding the integration of mutation testing into a development workflow.

Challenge 1: Mutation testing frameworks are often immature, and may have nuanced and error-prone behavior.

Recommendation 1: Assess the framework on a simple example to understand the framework and its limitations before integrating it into the full codebase.

The first recommendation is to develop a proof-of-concept to evaluate a candidate mutation testing framework’s features, nuances, workflow, and limitations. This enables developers to observe and develop solutions for errors and incompatibilities with the codebase before attempting a full integration. Not all issues will be discovered, but many will—and the issues encountered will be easier to debug in a simplified environment.

Challenge 2: Issues can emerge from the combination of the mutation testing framework, codebase, and/or build system.

Recommendation 2: When planning, include time for investigating and handling issues emerging from this integration.

For example, we encountered an issue where the build system compiled code in a sandbox, then moved files outside. It is difficult to predict integration issues in advance as they will differ between mutation testing frameworks, codebases, and build systems, so project planning must include time to debug these issues.

Challenge 3: Integrating a mutation testing framework requires project and build system expertise.

Recommendation 3: Rather than prioritizing mutation testing experts, involve developers who have expertise in the codebase, build systems, and tool maintenance during the integration process.

We found that little knowledge of mutation testing was required during integration. Rather, general experience with tool integration, as well as specific experience with the codebase and build system, were more important. Involving developers with such expertise, even if they are not normally involved in testing, will help ensure a smooth integration.

Challenge 4: Issues that emerged when the integration was moved from a local environment to CI were difficult and time-consuming to debug.

Recommendation 4: When integrating a mutation testing framework, perform the integration incrementally, both locally and in CI.

When performing the initial integration in a local environment, we operated in a manner where test suites and mutation operators were gradually added. This enabled a quick feedback loop when investigating and correcting issues. However, we then moved our integration into the CI pipeline all at once. The new issues that emerged required a significant amount of time to investigate and fix due to the increased feedback time and complexity. We recommend adopting an incremental integration in both environments.

Challenge 5: A significant amount of code needed to be developed to integrate the mutation testing framework.

Recommendation 5: Framework developers should prioritize the flexibility of their mutation testing frameworks, so that

they can be deployed for a variety of use cases and software ecosystems.

Vercammen et al. found that “mature testing tools that break down the initial startup effort and continuous human effort cost are needed before companies will be willing to integrate mutation testing in their workflows. [35]” Our findings are in line with this statement. Mull is one of the most mature C++ mutation testing frameworks [25]. Yet, we still needed to dedicate a significant amount of time to adapt it for our specific use case, codebase, and build system. As is, we consider the integration of mutation testing to still be problematic and would expect many developers to be dissuaded from attempting it or to give up.

Ideally, integrating a new tool should not require significant adaptation effort, and the need for such effort will slow adoption of new techniques. Further, the need for maintenance introduces an ongoing cost within an organization. The developers of mutation testing frameworks should focus usability and flexibility, which would lower the initial integration cost and ease maintenance.

It is unrealistic to expect developers to invest in mutation testing unless there is clear evidence that it will provide such value. The report and the dashboard were both positively received. If the initial positive impressions continue, it is likely that the integration will be fixed when it breaks. However, if not, it will slowly deprecate. We consider the regular use of mutation testing results by developers to be an essential part of a successful integration process.

7.2 Dashboard and Report (RQ2–3)

Participants offered insight into how they would use mutation testing, who should use it, what information they wanted, how it should be presented, and how beneficial the results were.

Finding 1: Providing an overview of mutation testing results in the dashboard enabled developers to more easily identify the areas most in need of improvement in the test suite.

Recommendation 6: The ability to gain an overview of mutation testing results is important, not just the ability to see surviving mutants.

When discussing mutation testing, there is often a focus on the specific surviving mutants. However, the participants also found value in using mutation testing to gain an overview of the test suite. There should be more research on how to efficiently use mutation testing to judge the health of a test suite over time and to identify under-tested aspects of the software. The dashboard, by offering filtering and visualizations, made it easier to gain this overview than a situation where developers had the report alone.

Finding 2: Mutation testing can benefit non-technical stakeholders when presented in a form that does not require knowledge of the codebase.

Recommendation 7: Certain mutation testing information (e.g., filterable mutation scores), overviews, and visualizations can offer non-technical stakeholders observability into the testing process.

Mutation testing is framed as a tool for developers who directly work on the codebase-under-test. Our observations suggest that mutation testing can also be used by non-technical stakeholders (e.g., team managers) to gain observability into the testing process via the visualisations in our dashboard.

Finding 3: Developers were primarily interested in mutation score and surviving mutants. They were not interested in mutation operator details or raw number of surviving mutants.

Recommendation 8: When presenting mutation testing results to developers, emphasise the mutation score and details on the surviving mutants.

Mutation testing frameworks can present multiple forms of information after executing the test suites. We found that the primary pieces of information valued by developers were the mutation score and details on the specific mutations that survived testing (e.g., location and the change made). Other information, such as the number of mutants created per operator, the list of operators applied, or the raw number of surviving mutants were not seen as useful.

Finding 4: Developers desire more contextual information about surviving mutants to make the results more actionable.

Recommendation 9: Combine mutation testing information with other data, such as code coverage or complexity.

When asked what information was missing to make mutation testing useful for improving test quality, participants consistently requested contextual information related to the mutated aspects of the code—e.g., establishing traceability between mutations and code coverage, source code metrics (such as cyclomatic complexity) for the areas of the code with surviving mutants, and the safety standards that the company must meet.

In current research, mutation testing is generally applied alone. However, linking mutation testing results with measurements from other testing and monitoring tools integrated into a CI pipeline can offer a more effective path to improving test and code quality.

Finding 5: Developers desire information to help prioritize which surviving mutants to target for elimination.

Recommendation 10: Frameworks should provide contextual information related to the severity of surviving mutants.

Recommendation 11: Develop guidance and link mutation testing results to other information and tool results to enable developers to prioritize surviving mutants.

Another observation—also discussed in related work [4, 28]—is that developers struggle to prioritize surviving mutants for elimination. Participants requested a way to establish the “severity” of each mutant and the effort that would be required to eliminate each. Future research should explore how to estimate the severity and effort, including both general and domain-specific factors for prioritization. Researchers and development organizations should work to establish appropriate guidance, and the developers of mutation testing frameworks should consider incorporating information that could help developers make prioritization decisions (e.g., code quality and complexity measurements).

Finding 6: Visualizations enable developers to understand the evolution of test quality over time.

Recommendation 12: Accompany mutation testing results from a single execution with visualizations showing the evolution of the results from past executions.

The reports generated by current mutation testing frameworks reflect the results of a single execution of the framework. One of the most useful aspects of the dashboard, as pointed out by participants, was the ability to understand how mutation testing results have changed over time. Such visualizations are important in giving observability into the current state of the testing process.

Finding 7: Participants found scalability and simplicity to be important when interpreting visualizations.

Recommendation 13: Focus on simple visualizations, filtered to a chosen level of granularity, over complex visualizations showing multiple levels of a hierarchy simultaneously.

Visualizations that are difficult to interpret can hinder the usefulness of mutation testing results. The participants favored simple visualizations, largely rooted in the current level of granularity—team, directory, or single file—to be the easiest to interpret and use, while hierarchical visualization such as tree maps were seen as harder to interpret.

Finding 8: Mutation testing information and visualization can be presented at multiple levels of granularity (team, directory, file), each enabling different use cases for stakeholders. The ability to filter results to a level improves the efficiency and effectiveness of mutation testing.

Recommendation 14: Allow filtering information and visualizations to different levels of granularity.

Different mutation testing information and visualizations can be presented based on different levels of result granularity, including the team, directory, and file levels. Each level is useful for different use cases, at different times and to different stakeholders—e.g., a team manager may use the “team” level to understand how test quality is evolving over time, while a specific developer may look at surviving mutants in a single file.

The ability to filter to a specific level and only see relevant information and visualizations enables more efficient and effective improvement of test quality and improves the usability of mutation testing results. Future approaches should include both the ability to filter, as well as the ability to directly navigate between visualizations and relevant aspects of the report.

Finding 9: Many participants were unfamiliar with mutation testing and required guidance to use the solution successfully.

Recommendation 15: Education (e.g., workshops or user guides) is still needed before introducing mutation testing into a development workflow.

Finally, it should be highlighted that mutation testing is still not a widespread technique. Participants reflected that they still needed to use the solution more before they could understand what a “good”

mutation score was. Improved mutation testing frameworks are not enough to ensure the technique is adopted—education and guidance are needed before mutation testing can offer value.

7.3 Threats to Validity

Conclusion Validity: The sessions were performed with a relatively small number of participants. However, during the thematic analysis, saturation was reached in fewer than 10 sessions.

Internal Validity: We answered RQ1 based on our own observations of an integration process that we performed, introducing further risk of biased interpretations. We attempted to mitigate this threat by following a systematic observation protocol. Furthermore, developers at Zenseact also participated in making and discussing observations, reducing bias.

Participants may have given incomplete or inaccurate answers during the think-aloud and interview sessions since they were being observed or to appear more agreeable with researchers. Resistance to change and biases regarding organizational culture are also possible. Such biases are an expected risk during qualitative analyses, and are mitigated by focusing on common themes—not individual statements [36]. Agreement between coders was also assessed and used to strengthen the analysis.

Construct Validity: Mutation testing is still relatively uncommon, and participants may have misunderstood underlying concepts. We gave an overview of mutation testing to mitigate this threat. Participants could also ask for clarification at any time.

External Validity: The case study was conducted at a single company, based on code in C++ and a CI pipeline based on the Bazel build system. We also focused on a single mutation testing framework, Mull. All of these factors potentially limit the generalizability of our findings. However, we argue that our findings are not specific to the build system, language, or framework. We believe that our findings will apply, at a minimum, to similar contexts, such as organizations developing safety-critical systems.

8 CONCLUSION

In this study, we have explored the *technical* challenges of implementing mutation testing in continuous integration, *what* information from mutation testing is of use to developers, and *how* that information should be presented.

Ultimately, we have identified five technical challenges, including tool immaturity, issues emerging from the combination of mutation testing framework, codebase, and build system, and the integration process itself. We also offer 16 recommendations regarding mutation testing information including the importance of gaining an overview of test suite quality and its evolution, the information useful to different stakeholders, missing contextual information, the value of filtering mutation testing results for different levels of granularity, effective visualization, and the continuing need for education on how to interpret mutation testing results.

In future work, we aim to investigate different metrics that could offer context lacking in current mutation frameworks. We will also implement additional visualizations, validate our findings at additional organizations, explore how mutation testing can be used by different stakeholders, and conduct a long-term study to quantify the impact of mutation testing over time.

REFERENCES

- [1] Yoram Adler, Noam Behar, Orna Raz, Onn Shehory, Nadav Steindler, Shmuel Ur, and Aviad Zlotnick. 2011. Code coverage analysis in practice for large systems. In *Proceedings of the 33rd International Conference on Software Engineering*. 736–745.
- [2] Mauricio Aniche. 2022. *Effective Software Testing: A developer's guide*. Simon and Schuster.
- [3] Sebastian Baltes and Paul Ralph. 2022. Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering* 27, 4 (2022), 94.
- [4] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What it would take to use mutation testing in industry—a study at facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 268–277.
- [5] Joakim Brännström. [n.d.]. Dextool Mutate. <https://github.com/joakim-brannstrom/dextool/>.
- [6] Yiqun T Chen, Rahul Gopinath, Anita Tadakamalla, Michael D Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. 2020. Revisiting the relationship between fault detection, test adequacy criteria, and test set size. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*. 237–249.
- [7] ISO/TC 22/SC 32 Committee. 2018. *Road vehicles — Functional safety* (2 ed.). Standard. International Organization for Standardization, Geneva, CH.
- [8] Renzo Degiovanni and Mike Papadakis. 2022. μ BERT: Mutation Testing using Pre-Trained Language Models. arXiv:2203.03289 [cs.SE]
- [9] A. Denisov and S. Pankevich. [n.d.]. Mull. <https://github.com/mull-project/mull/>.
- [10] Martin Fowler. 2006. Code Ownership. <https://martinfowler.com/bliki/CodeOwnership.html>. Accessed: 2024-03-21.
- [11] Dawson R Hancock, Bob Algozzine, and Jae Hoon Lim. 2021. *Doing case study research: A practical guide for beginning researchers*. Teachers College Press. 47–50 pages.
- [12] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *FSE 2014, Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*. Hong Kong, 654–665.
- [13] Mingwan Kim, Neunghoe Kim, and Hoh Peter In. 2020. Investigating the relationship between mutants and real faults with respect to mutated code. *International Journal of Software Engineering and Knowledge Engineering* 30, 08 (2020), 1119–1137.
- [14] Klaus Krippendorff. 2018. *Content analysis: An introduction to its methodology*. Sage publications. 221–222, 251–252 pages.
- [15] Klaus Krogmann, Christian M Schweda, Sabine Buckl, Michael Kuperberg, Anne Martens, and Florian Matthes. 2009. Improved feedback for architectural performance prediction using software cartography visualizations. In *Architectures for Adaptive Software Systems: 5th International Conference on the Quality of Software Architectures, QoSA 2009, East Stroudsburg, PA, USA, June 24–26, 2009 Proceedings* 5. Springer, 52–69.
- [16] Nan Li, Michael West, Anthony Escalona, and Vinicius HS Durelli. 2015. Mutation testing in practice using ruby. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 1–6.
- [17] Yang Li and Walid Maalej. 2012. Which traceability visualization is suitable in this context? a comparative study. In *Requirements Engineering: Foundation for Software Quality: 18th International Working Conference, REFSQ 2012, Essen, Germany, March 19–22, 2012. Proceedings* 18. Springer, 194–210.
- [18] Wei Ma, Thomas Laurent, Miloš Ojdanić, Thierry Titchou Chekam, Anthony Ventresque, and Mike Papadakis. 2020. Commit-aware mutation testing. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 394–405.
- [19] Wei Ma, Thierry Titchou Chekam, Mike Papadakis, and Mark Harman. 2021. Mudelta: Delta-oriented mutation testing at commit time. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (2021). <https://doi.org/10.1109/icse43902.2021.00086>
- [20] Stryker Mutator. 2024. Mutation Testing Elements Schema. <https://github.com/stryker-mutator/mutation-testing-elements>. Accessed: 2024-06-14.
- [21] Elastic NV. 2024. Kibana. <https://www.elastic.co/kibana>. Accessed: 2024-03-25.
- [22] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology* 5, 2 (1996), 99–118. <https://doi.org/10.1145/227607.227610>
- [23] Milos Ojdanic, Ezekiel Soremekun, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. 2023. Mutation Testing in Evolving Systems: Studying the relevance of mutants to code evolution. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–39.
- [24] Sebastiaan Oosterwaal, Arie van Deursen, Roberta Coelho, Anand Ashok Sawant, and Alberto Bacchelli. 2016. Visualizing code and coverage changes for code review. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 1038–1041.
- [25] Jonathan Örgård, Gregory Gay, Francisco Gomes de Oliveira Neto, and Kim Viggedal. 2023. Mutation Testing in Continuous Integration: An Exploratory Industrial Case Study. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 324–333.
- [26] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.
- [27] Ali Parsai and Serge Demeyer. 2020. Comparing mutation coverage against branch coverage in an industrial setting. *International Journal on Software Tools for Technology Transfer* 22, 4 (2020), 365–388.
- [28] Goran Petrovic, Marko Ivankovic, Bob Kurtz, Paul Ammann, and René Just. 2018. An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 47–53. <https://doi.org/10.1109/ICSTW.2018.00027>
- [29] M. Pezze and M. Young. 2006. *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons.
- [30] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. 2023. Robust speech recognition via large-scale weak supervision. In *International Conference on Machine Learning*. PMLR, 28492–28518.
- [31] Marvin Rausand. 2014. *Reliability of safety-critical systems: theory and applications*. John Wiley & Sons.
- [32] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering* 14 (2009), 131–164.
- [33] Gayane Sedrakyan, Erik Mannens, and Katrien Verbert. 2019. Guiding the choice of learning dashboard visualizations: Linking dashboard design and data visualization concepts. *Journal of Computer Languages* 50 (2019), 19–38.
- [34] Edward R Tufte. 2001. *The visual display of quantitative information*. Vol. 2. Graphics press Cheshire, CT.
- [35] Sten Vercacmmen, Markus Borg, and Serge Demeyer. 2023. Validation of Mutation Testing in the Safety Critical Industry through a Pilot Study. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 334–343.
- [36] Victoria Clarke Virginia Braun. 2021. *Thematic Analysis - A practical guide* (1st ed.). SAGE Publications Ltd.
- [37] Zenseact. 2024. Zenseact LinkedIn Post. https://www.linkedin.com/posts/zenseact_today-eid-al-fitr-marks-the-end-of-ramadan-activity-7183796792193413121-FEUW?utm_source=share&utm_medium=member_desktop. Accessed: 2024-04-16.
- [38] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. 2019. Predictive Mutation Testing. *IEEE Transactions on Software Engineering* 45, 9 (2019), 898–918. <https://doi.org/10.1109/TSE.2018.2809496>