CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Mutation Testing in Industry CI: A Value-Centric Approach

A case study about the intersection of software quality and developer experience.

Master's thesis in Computer science and engineering

STEFAN ALEXANDER VAN HEIJNINGEN, THEO WIIK

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Mutation Testing in Industry CI: A Value-Centric Approach

A case study about the intersection of software quality and developer experience.

STEFAN ALEXANDER VAN HEIJNINGEN, THEO WIIK

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Mutation Testing in Industry CI: A Value-Centric Approach
A case study about the intersection of software quality and developer experience.
STEFAN ALEXANDER VAN HEIJNINGEN, THEO WIIK

Supervisor 1: Gregory Gay, Department of Computer Science and Engineering
Supervisor 2: Francisco Gomez, Department of Computer Science and Engineering
Advisor 1: Kim Viggedal, Zenseact
Advisor 2: David Friberg, Zenseact
Examiner: Richard Torkar, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Mutation Testing in Industry CI: A Value-Centric Approach
A case study about the intersection of software quality and developer experience.
STEFAN ALEXANDER VAN HEIJNINGEN, THEO WIIK
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Context: Mutation testing is a robust testing technique to assess the sufficiency of test suites. The industry is still struggling to adopt mutation testing and maximize its benefits despite recent research suggesting it is becoming more mature.
Objectives: This study aims to share insights and recommendations to:

1. Assist developers during the integration process of mutation testing tools.
2. Present mutation testing results to maximize their benefit and minimize the cost of using them.

Methods: We perform a case study in an industry setting. We create an experience report that reflects on the integration process of a mutation testing tool at the partner company. We then focus on developer experience by using ten think-aloud sessions and semi-structured interviews to explore what information developers perceive as useful and how the information should be presented.
Results: A CI pipeline was developed to run mutation testing nightly and upload results to a developed dashboard. Integrating mutation testing tools is still a challenging process. When interacting with results, developers valued interactivity, getting an overview, and wanted to associate mutation testing results with other contextual information of the codebase, such as code coverage and complexity. A set of recommendations was created to facilitate integrating and using mutation testing in industry settings.
Conclusion: As-is, mutation testing is a standalone tool that should be easier to integrate and interact with other aspects of the codebase to become more adopted and successful.

Keywords: software engineering, quality assurance, test sufficiency criteria, mutation testing, continuous integration, developer experience, case study, thesis

Acknowledgements

First and foremost, we would like to thank our academic supervisors Gregory Gay and Francisco Gomes for their guidance and invaluable feedback throughout our thesis.

We also want to thank our company supervisors David Friberg and Kim Viggedal for the support and guidance that they have provided us.

From Zenseact, we furthermore also want to thank Mats Nilsson for the technical support he has given us, and all of the developers that participated in our study.

Stefan Alexander van Heijningen & Theo Wiik, Gothenburg, June 2024

Contents

List of Figures

List of Tables

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis, listed in alphabetical order:

CC	Code Coverage
CI	Continuous Integration
LLVM IR	LLVM Internal Representation
LOC	Line(s) of Code
ML	Machine Learning
MT	Mutation Testing
SUT	System Under Test
VCS	Version Control System

1
Introduction

Ensuring software works as intended is crucial, especially in safety-critical systems where faults can lead to severe consequences. Test suites systematically assess that a software system behaves as expected, enabling the detection of faults and preventing adding new faults. To measure the sufficiency of a test suite, test-adequacy criteria — such as line coverage — can be used to verify that the test suite tests the system appropriately. Software developers use said criteria to check if their test suites are sufficient and improve those not [3].

Mutation coverage, obtained by Mutation Testing (MT), is considered one of the most robust test-adequacy criteria [4]. Mutation testing is a technique to evaluate test suites by purposely modifying the code's behavior and checking whether the test suite detects said modification by failing. All tests passing even though the code's behavior was modified is considered a flaw in the test suite.

However, MT is not widely adopted in the industry due to the computational costs and the lack of maturity of MT tools [5]. Recently, there has been a significant effort to reduce the computational costs of MT. At the same time, the availability of open-source and industrial tools has surged, which suggests that MT is reaching a mature state [4, 6]. The same is implied by recent research [7, 8, 9, 10, 11], which is more commonly being done in industry settings, suggesting that industry is willing to start adopting MT.

Another test-adequacy criterion is Code Coverage (CC), which measures how much source code is executed when running the test suite. Code coverage measures have shortcomings [12]; for instance, code coverage does not reflect whether the tests validate the intended behavior of the System Under Test (SUT), nor whether the tests consider edge cases or boundary values. Nevertheless, industrial settings often only use CC as a criterion [13] and could benefit from more robust test-adequacy criteria such as MT, which could be complementary to CC [5].

As argued in Section 1.1, cost and lack of maturity are not the only factors holding back MT from being integrated into the industry: there is also a lack of guidelines and best practices to successfully adopt and utilize MT in a way that minimizes its integration and usage costs while maximizing its benefits. To address this lack of best practices, guidelines, and industry adoption, we performed a case study at

Zenseact[1], a company within the automotive industry that uses C++ in a safety-critical setting. The case study had two focuses: the technical hurdles typically encountered when adopting MT and the format in which MT results are presented to developers.

In the case study, we first integrated the MT tool Mull[2] into the company's build system and Continuous Integration (CI) pipelines to run MT nightly on the company's codebase and then present results in a Kibana dashboard[3]. Observations were collected during the integration, which were then used to synthesize an experience report and a set of recommendations regarding the integration process of an MT tool. We then conducted think-aloud sessions and interviews with 10 developers from the partner company. During these activities, we presented the Kibana dashboard and an HTML report to developers to explore various methods of presenting the MT results to the developers. The aim was to convey the MT results in a way developers perceive as useful for improving their test suite without causing issues such as information overload.

Overall, the study aims to provide recommendations for utilizing MT in an industry setting and evaluate what and how to convey the data to the developers. This effort aims to support the industry in adopting MT and fill the relatively little-researched area of utilizing the generated mutants in a way developers find beneficial.

1.1 Problem Statement

The vast majority of research related to MT still focuses on reducing the computational cost. Nevertheless, some research includes qualitative analysis of developers' experiences in industry settings when they attempted to introduce MT. Despite its potential, the current literature reveals a trend: a struggle to adopt MT into developers' workflow when using the traditional way to work with MT results.

For instance, Petrovic et al. [14] argue that adopting MT is challenging with the traditional workflow of MT. Beller et al. [15] reflect that most developers in their study were too unfamiliar with MT to use it effectively. Vercacmmen et al. [16] performed a study with two different companies, where one company struggled to use MT effectively, while the other reflected on issues they had to address to make MT successful for them. The common pattern that can be observed reveals the following problem:

> The computational cost is no longer the only factor holding back the deployment of mutation testing in the industry. The lack of best practices and guidelines on integrating and using mutation testing is also a significant factor that deters developers from embracing mutation testing in the industry.

[1] `https://zenseact.com/`
[2] `https://github.com/mull-project/mull`
[3] `https://www.elastic.co/kibana/kibana-dashboard`

The problem above has been relatively unexplored within MT research, leading to a lack of empirically validated best practices and guidelines that development teams could use to integrate MT into their workflow as seamlessly as possible to facilitate getting additional test-adequacy criteria in the form of mutation coverage and increase the developer's perceived usefulness and use of MT.

There is research that focuses on presenting the most useful mutants to the developer through techniques such as filtering and classification. Although this enables easier adoption of MT in the industry, the techniques presented are not easily adaptable by other companies that wish to start using MT for several reasons. For instance, the techniques presented tend to be implemented for a specific tool, meaning they are unusable if other companies aren't using the same tools. Regardless, the implementations of the methods are typically not accessible and are instead privately used by the companies that developed them. Techniques also tend to depend on specific programming languages [4] and typically involve training a machine learning model that decides what a helpful mutant is. The amount of effort a company would have to put in to develop a similar system is excessive and unrealistic, especially for companies only looking to try out MT.

Due to the previously described issues, most techniques that facilitate adoption are not easily transferrable to companies that only wish to experiment with applying MT to their code base for the first time to see whether they want to adopt it. More applicable recommendations can aid those who might otherwise struggle when attempting to adopt MT, by improving the developer experience and providing value early on. To make the recommendations developed during this research more applicable to different settings, they aim to be agnostic to the specific language, tool, or training data used.

1.2 Research Objective

This study aims to make MT a viable option in the industry by providing developers with insights, information, and recommendations. To do so, we first share insights when integrating MT at the partner company, reflecting on the process and the challenges faced. Based on our experience, we provide recommendations for integrating MT into a codebase, which can aid those wishing to integrate MT into their own development workflow.

We then evaluate information from MT results with developers at the partner company to explore what they perceive as useful. We further evaluate how to present the information effectively so that it can be used to improve a test suite without issues such as information overload. By sharing what is perceived as the most useful information from MT in an interpretable way, we assist those wishing to maximize the amount of benefit they obtain from MT.

Our insights also aim to assist MT tool developers by guiding their design decisions, facilitating the integration of their tools, and generating information valued by tool

users.

The study's initial focus was on the challenges of applying MT in CI and then investigating how to present MT results. However, during the integration of MT tools, we realized that adopting MT requires a more holistic viewpoint, not just focusing on CI but also factors such as the build system, environment, maintainability, and ease of use for developers. We considered that we could provide more insightful information by not limiting ourselves to reflecting on CI and thus adjusted the scope of our study accordingly.

1.3 Significance of the Study

The practical contributions of this study are insights from an experience report and a set of recommendations for integrating MT in an industry setting. We furthermore indicate what MT information was perceived most as useful to developers, and how said information should be presented. Additionally, we present a developed dashboard for visualizing MT results in a novel way. These contributions aid developers in adopting MT by guiding the integration of MT tools, and how to provide value by using the MT tools.

Our scientific contributions include providing insights into the current state-of-the-practice usage of MT in industry settings. We share our experience of integrating MT, which shows what improvements MT tool creators should focus on. We furthermore highlight the need to focus on a new area in MT research: the developer experience. We evaluate the perceived usefulness of MT results presented using visualization techniques and share the shortcomings that could be addressed by future research. Future research directions are also proposed which other researchers can consider exploring.

1.4 Scope Delimitation

This study does not include a quantitative study measuring whether specific information or a visualization technique increases the test suite's quality. Instead, it focuses on the developers' experience when presenting the results of MT, focusing on information and visualization. A longitudinal study would be required to perform a quantitative analysis where actual benefits are measured, which is out of the scope of this study. Furthermore, our study does not focus on evaluating cost-reduction techniques or applying them. Some cost-reduction techniques were applied to make MT feasible at our host company; however, our research addresses the practical usefulness of MT results, so the cost of obtaining said results is not the main concern.

2

Background

This chapter explains the concepts relevant to the study. First, the concept of CI is introduced, along with why it is relevant for MT. The concept of Code Coverage Criteria is then described, exposing some of its flaws to highlight the need for MT. Mutation Testing is then explained in more depth, discussing how MT can be used in CI, the MT tool used during this project, and how MT results are typically presented. Last, we briefly cover how visualization techniques can convey information.

2.1 Continuous Integration

Continuous integration incorporates practices coined by Fowler and Foemmel [17] to minimize the amount of work required to integrate new changes into a codebase. The code is continuously built and tested once a change is added to the Version Control System (VCS) in a remote environment. With CI, developers are encouraged to commit often, minimizing the cost of merging their commits into the repository. Continuous integration has several benefits, such as continuous feedback or allowing errors to be caught early. As a disadvantage, however, CI requires maintaining more infrastructure and dependencies.

It also replaces manual ad-hoc procedures, automating processes and thus enabling developers to dedicate their time elsewhere. The procedures can include calculating the line coverage or performing static code analysis with tools such as SonarQube[1] and then uploading the results. These results are then stored in an accessible location to developers. Most modern DevOps platforms offer solutions for running automated scripts in pipelines, with tools such as GitHub Actions[2] or Azure Pipelines[3].

Since CI reduces the time developers dedicate to processes, it's fitting to MT, something already being done as mentioned in Section 3.1. By automating MT in CI, the amount of time a developer has to dedicate to MT diminishes, as they only have to look at the results instead of obtaining them.

[1] `https://www.sonarsource.com/products/sonarqube/`
[2] `https://github.com/features/actions`
[3] `https://azure.microsoft.com/en-us/products/devops/pipelines/`

2.2 Code Coverage Criteria

Code coverage is a group of measurements used to determine the amount of code executed when running a test suite's tests. The goal is to obtain a quantitative measure of the test suite's source code coverage, which can be interpreted. Different CC measures can be used, such as line, branch, condition, path, and MC/DC coverage. The various options differ in the amount of tool support, computational cost, and comprehensiveness. A commonly used metric from CC is line coverage (the percentage of lines of code executed). Usually, context-dependent thresholds are set as goals for teams, and once said threshold is reached, the test suite is deemed sufficient.

However, reaching a certain threshold does not necessarily mean that edge cases are being tested, and a test suite can have a high CC percentage while only testing a limited amount of scenarios that the software system could encounter. Furthermore, CC does not reflect whether the tests verify the behavior of the source code, only whether it is executed or not. A pseudocode example can be seen in Listing 2.1. The source code is a simple function tested by two unit tests. The two tests together obtain 100% line coverage, misleadingly informing developers that the tests are sufficient, even though the tests do not cover values that are prone to faulty behavior, such as boundary values. As a consequence, future changes that break the intended behavior of the code could be undetected by the tests.

```
def can_drive(age: int) -> bool:
    return age >= 18

# Test 1: age < 18
def test_kids_cant_drive():
    assert_false(can_drive(17))

# Test 2: age == 18
# missing

# Test 3: age > 18
def test_adults_can_drive():
    assert_true(can_drive(19))
```

Listing 2.1: Pseudocode code missing a test for a boundary case whilst still having 100% line coverage.

2.3 Mutation Testing

Mutation testing is a white-box test-sufficiency technique that evaluates test suites. To do so, code changes are introduced into the source code, modifying its original behavior. Then, the test suite is executed to see whether the modified behavior is detected.

Each source code modification is known as a *mutant*. There are two possibilities when a test suite is executed for a mutant:

- The test suite does not detect the mutant because all tests still pass. In this case, the mutant is considered a *surviving* mutant.
- At least one test fails, effectively detecting the changed behavior caused by the mutant. In this case, the mutant is considered a *killed* mutant.

Mutation operators determine which source code to modify and how, for example, by replacing an arithmetic expression (e.g., a + to −, see Table A.1). Normally, developers can configure which mutation operators to enable when using MT.

To perform MT, automated tools are used. The tools use the mutation operators enabled by the developer to scan for possible mutants in the source code. Then, the test suite is executed for each mutant, and the outcome of the tests is checked to determine whether the mutant survived or was killed. It should be noted that the exact behavior of different MT tools can vary and can frequently be configured.

Once the tool has finished, it traditionally shows two types of results. The first result is the *surviving mutants*, mentioning which modification was made in the source code for each one. The second result is the *mutation score*, which is computed as $^{\text{killed}}/_{\text{total}}$ mutants. The mutation score is comparable to CC scores; the higher, the better. Like CC, context is used to interpret the resulting value, and frequently, a context-dependent threshold is set to determine whether tests are sufficiently tested. A system operating in a safety-critical environment will likely have a higher mutation score threshold than a system with less severe consequences in the case of a failure.

Mutation testing is considered very advantageous in addition to CC, as it verifies whether tests validate the behavior of source code instead of just indicating how much code the tests execute, as CC does. This makes it harder to increase the metric artificially by only testing the happy paths to cover as many lines of code as possible.

However, MT also has its issues. Traditionally, the most significant issue holding back MT in the industry has been its computational cost. First, the code has to be scanned for places that can be mutated. Then, the code must be compiled (when applicable), and the test suite runs once per mutant.

There are also issues related to the results generated by MT. Equivalent mutants are mutants that do not modify the behavior of the source code. Since the behavior isn't modified, it is considered that the mutant survived, which negatively impacts the mutation score misleadingly. Detecting equivalent mutants is challenging since it requires knowledge about the source code, the tests, and an interpretation of what occurs during execution when the mutant is inserted. Undefined behavior mutants is an issue related to MT in programming languages such as C++, where the modification in the source code makes it break the language rules [18]. This causes unpredictable and flaky behavior. However, the issues mentioned above are not the focus of this research.

2.3.1 Using MT in CI

When it comes to using MT, different techniques can be used, each with its own advantages and disadvantages. We see the following ways in which MT can be used (similar to the findings by Örgård et al. [19]):

Manual This strategy consists of the developer running MT in a manual and ad-hoc fashion, without setting it up in the CI. Once the developer has finished interpreting MT results, they might modify or add the unit tests. To see the impact of their changes, they would need to start the whole process over again, which is repetitive.

Per change In this strategy, MT is applied to a modified piece of code. This technique is useful for code reviewing, as it allows observing how the modified code impacts the test sufficiency. However, it is computationally heavy due to the frequency of commits being added and thus has a long feedback loop. Furthermore, support from the MT tool is required to enable this strategy.

Periodically Mutation testing can be applied periodically when computational resources are available, such as nightly or during the weekend. This technique circumvents problems related to the computational cost of MT and provides a more holistic view of the code sufficiency of a codebase. A disadvantage, however, is that new results are only available after each CI run, increasing the feedback loop.

Mull[4] is an MT tool for C and C++ validated as a viable MT option for CI in previous work by Örgård et al. [19]. Their findings highlight that, compared to other tools, Mull was easy to install, had the required features needed for CI, had the essential mutation operators, and was relatively fast. However, it does not support applying MT to multiple test suites simultaneously. Their study also created a proof of concept where they used Mull in a CI pipeline.

The tool performs MT in two steps. The first step consists of compilation, during which the tool scans for opportunities to inject mutants. Each mutant is activated and deactivated with execution flags. Due to the execution flags, only one compilation is required for all mutants, making the tool faster than similar tools. The second step consists of execution, where each mutant is activated, and the complete test suite is executed. The tool checks whether any test failed or not. Once all mutants have been checked, a report is generated. Mull offers different report formats, such as HTML, JSON, and SQL.

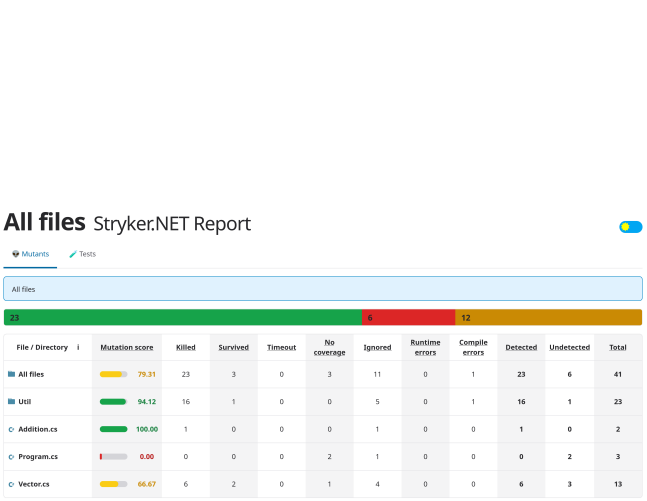
2.3.2 Mutation Testing Reports

Traditionally, MT tools present their results by generating reports in different formats, such as JSON, XML, or HTML. These reports generally contain information such as what mutation operator was applied, where in the code (column and row),

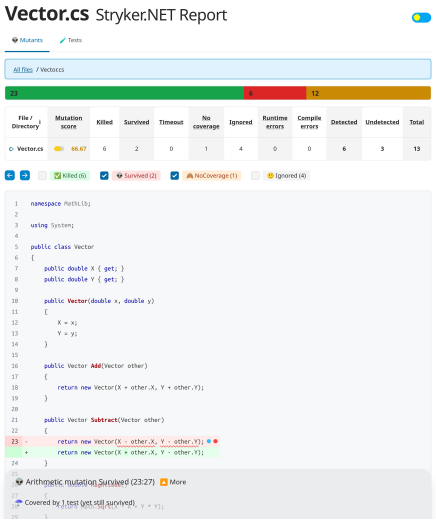
[4] `https://github.com/mull-project/mull`

and whether the mutant was killed or survived.

An example of a report with a corresponding open-source schema for MT results is from the Stryker Mutator project[5]. Stryker Mutator is a project with MT tools for JavaScript, C#, and Scala, which all share the same HTML reporter enabled by the schema (Mutation Testing Elements[6]). This reporter is also used by Mull, thus producing the same type of HTML reports. In Figure 2.1a, an example of an HTML report used by Stryker Mutator and Mull can be seen. The example shows MT concepts such as survived mutants, killed mutants, and mutation score. Directories can be navigated in the report, and individual files can be opened to see the surviving mutants in the source code. For an example, see Figure 2.1b.



(a) Example HTML report viewing file overview.



(b) Example HTML report viewing surviving mutants in a file.

Figure 2.1: Example of HTML reports generated by Stryker Mutator.

HTML reports from popular MT tools such as Mull (C++), mutmut (Python), Pitest (Java), and Stryker Mutator (C#) all report similarly. They have an overview showing the mutation score per file and directory. Users can click on files and observe surviving mutants in the source code.

2.4 Visualization of Data

Visualization is a technique for displaying information graphically to facilitate its interpretation, analysis, and actionability. We argue that using efficient visualization techniques is important for presenting data from tools conveniently, as otherwise, developers are less inclined to use the tool. Information visualization is common

[5] `https://stryker-mutator.io/`
[6] `https://github.com/stryker-mutator/mutation-testing-elements`

in software development and can show results related to CC, code complexity, test execution, requirement traceability, performance testing, MT, etc.

However, visualization of large amounts of complex data is challenging. Tufte [20] shares many principles to achieve clarity, precision, and efficiency when visualizing information. He argues that all clutter should be removed from visualizations, thus presenting as much information as possible using a minimum amount of visual elements, which he calls maximizing the Data-ink ratio. He furthermore argues about data density, stating that there are limits to the amount of information that can be displayed before a visualization becomes overwhelming.

Kibana is a dashboard tool developed by Elastic[7] and is used to present mutation testing results in this study. Elastic offers services such as real-time and infrastructure log monitoring and profiling. Kibanas dashboard allows visualizations of said logs, which are useful for monitoring system health over time and in real-time. For an example, see Figure 2.2, which shows a dashboard of network logs. This dashboard presents metrics such as the peak hours of traffic with the bar chart at the top right or the number of visited pages in the table at the bottom left.

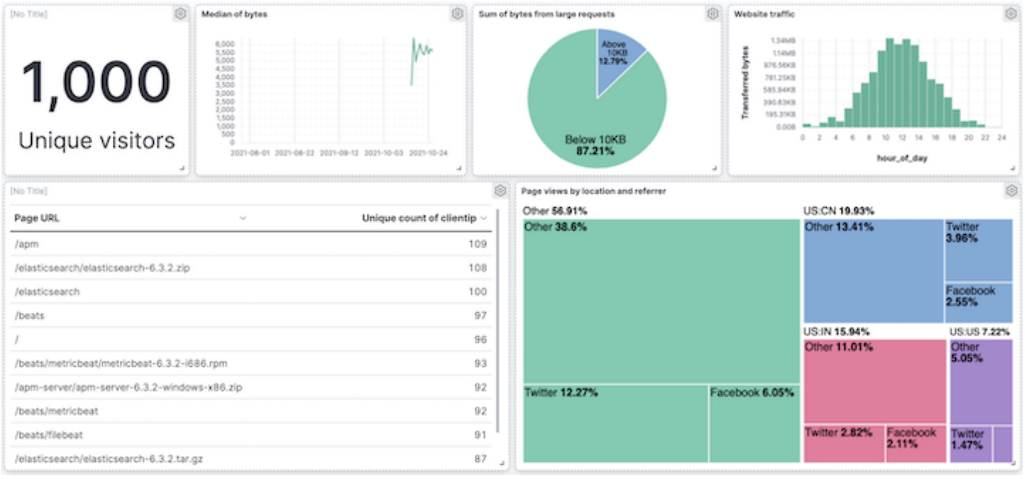


Figure 2.2: Example Kibana dashboard [1].

[7] https://www.elastic.co/

3

Related Work

There is a limited amount of literature in the field of MT strongly related to the challenges encountered when incorporating MT in an industry setting and about developer experience when using MT results. However, there is MT research that highlights other aspects yet provides useful insights, which is covered in the first section. Relevant research also exists in other fields whose insights and techniques can be adapted and evaluated for their potential when applied to MT. Therefore, this section will cover related research on MT, CC, and CI pipelines.

3.1 Mutation Testing

In the related literature, the research most pertinent is by Örgård et al. [19], which this research builds upon. It includes assessing existing C++ MT tools and their feasibility in CI/CD. The choice of MT tool used in this study is based on said assessment. Furthermore, developers were interviewed on how they would like to use MT in CI/CD. However, the results of both activities were not validated in an industry setting. Moreover, no reflection is made on the challenges faced when a company intends to adopt MT.

Much of MT research focuses on other aspects but shares insights into the adoption challenges encountered. Vercacmmen et al. [16], Petrovic et al. [14] and Beller et al. [15] all suggest that addressing every mutant in large projects is not feasible because of the time needed to analyze the surviving mutants and to create corresponding tests. However, to solve the issue, they employed techniques such as reducing the number of mutation operators used and mutants to analyze instead of experimenting with what information to present and how to present it.

Petrovic et al. [14] argue that the traditional usage of MT (i.e., observing generated reports and creating tests based on them) does not scale sufficiently when integrating MT into already existing large codebases, as there are too many mutants to analyze one by one in a reasonable timeframe. To make MT viable in their large codebase, they reduce the number of mutation operators used, hiding surviving mutants automatically flagged as unproductive and only creating mutants related to the changes in commits.

Beller et al. [15] reflect that most developers were too unfamiliar with MT to use it effectively. Furthermore, even though running MT on their software revealed deficits in their testing suite, only half of the developers were inclined to create new tests to improve the test suite.

Vercacmmen et al. [16] performed a study at two companies, one without experience in MT while the other had five years of experience. The company, without experience, struggled to use MT effectively, referring to issues such as information overload and the cost of maintaining the tool in their environment. On the other hand, the company with more experience shared reflections on what they had to do to make MT worthwhile, such as providing simple, interpretable, and valuable feedback since developers wouldn't use the tool otherwise. They had to design a workflow to use MT and improve the information conveyed to the developers.

There is also some related research that relates MT to CI. Ma et al. [8] perform a study where they apply commit-aware MT techniques in CI, although CI is not the main focus of the study. On the other hand, Parsai et al. [5] perform a study about the viability of MT in CI. Another contribution Parsai et al. make is the examples of visualizations of mutation scores aggregated per class, which helped reduce the information overload from MT results. Although the way the information is presented is not the main focus of their research, it already shows that some practices are being followed to reduce information overload when using MT. They do not explain why they performed the aggregation or present developer experiences with the aggregated data.

3.2 Code Coverage

Similarly to MT, CC quickly ends in information overload on large codebases [21]. However, in contrast to MT, CC has more research on techniques dedicated to preventing this issue. Various research has been performed on how to present CC comprehensibly to enable the developer to understand what parts of the code to improve the testing on, as otherwise, the developer is likely faced with information overload. Adler et al. [21] mention some of the problems in presenting CC data to the users, akin to those prevalent in MT. The authors cover two aspects that make CC challenging in large codebases: first, the technical aspects of introducing line coverage into a software suite and presenting the information meaningfully; secondly, comparing it to the 'needle in a haystack' problem. The researchers validated the applicability of a specific technique (substring hole analysis), which proved viable by grouping parts with low coverage together. These sections represent areas that need improvement and aggregate data to give an overview.

A study by Ivanković et al. [22] covers Google's internal CC techniques. This study provides insights into how CC can be implemented in large code bases. At Google, CC is completely automated, without requiring any effort from the developer to obtain the results. Furthermore, it is also integrated into their code reviewing tool (Critique) and their repository navigation tool (CodeSearch). While Critique is used

to see how a given commit affects the CC, CodeSearch provides more of an overview.

The study at Google also includes surveys of the perceived usefulness of the CC tools when doing certain tasks, such as authoring, reviewing, or browsing code changes. It should be noted, however, that the survey also included people in non-engineering roles and that at Google, CC is used voluntarily, where teams set their own thresholds or can opt out altogether. The survey indicated that 16% of participants viewed CC negatively; however, the reasons for this are not discussed.

3.3 Continuous Integration Pipelines

Continuous integration is widely adopted in the industry, and extensive research has been conducted about its implementation and benefits [23]. Elazhary et al. [23] explored the industry's usage of CI and its benefits and challenges. They highlight several benefits, such as catching errors early by self-testing the build, keeping everyone updated with the latest source code, and avoiding context switching by providing automatic project builds [23].

Furthermore, the findings indicate that the practices of CI defined by Fowler and Foemmel [17] are adapted and prioritized according to one's needs and values [23]. The authors present an example of this adaptability; the value of a specific test strategy is assessed based on its added values: *"... if integration testing did not provide actionable and informative results—the three organizations did not prioritize it"* [23]. This highlights the importance of perceived benefits for a tool to be used; even though a tool might be useful in theory (e.g., validated by research), in the end, a developer has to perceive it as being useful for it to be utilized.

Berner et al. [24] document lessons learned when integrating CC analysis and visualization tools in a project. These lessons include what results to expect, how to motivate developers, and developer behavior to avoid (such as avoiding competition by reaching high coverage rates). Although aimed at CC tools, these lessons also apply to MT and were considered during our study.

When it comes to conducting MT in CI specifically, it should be noted that it is something that has been explored in multiple studies already [4, 9, 14, 16, 19], focusing on aspects such as feasibility and using it for code reviews. As such, per-commit MT is commonly used in CI, and related studies usually focus on filtering out unproductive surviving mutants using techniques involving Machine Learning (ML).

4

Methods

The following chapter presents the research questions, the case, and the methods used during the study to answer the research questions. The research methods and procedures for the study adhere to an exploratory and interpretivist case study, following the terminology defined by Runeson et al. [25] and Baltes et al. [26]. An experience report, think-aloud sessions, interviews, and thematic analysis were used to address the research questions.

4.1 Research Questions

The research questions can be categorized into two main parts: one focusing on the process of integrating MT in a development workflow (**RQ1**) and two focusing on how MT can best be used in a CI setting to maximize its perceived benefit (**RQ2-3**).

> **RQ1**: What challenges arise when integrating and automating the usage of mutation testing tools, and how can they be addressed?
>
> **RQ2**: What information from mutation testing in continuous integration should be conveyed to developers to be perceived as useful and why?
>
> **RQ3**: How should information from mutation testing in continuous integration be presented to developers to be perceived as useful, and why?

Regarding RQ1, mutation testing typically relies on multiple tools (package managers, compilers, unit testing frameworks, ...). As such, introducing an MT tool can lead to friction, including dependency conflicts, maintenance efforts, or user dissatisfaction. This is furthermore aggravated when adding MT to CI pipelines. After identifying the barriers that hinder such integration, we aim to share our experience addressing these issues to aid future integrations at other companies. This RQ aims to assist developers in integrating MT at their companies.

For RQ2, the most important information presented in MT results is, typically, the mutation score and surviving mutants shown in the source code. We investigated whether other aspects, such as mutation operators and the quantity of

surviving mutants, are information that developers perceive as useful. Providing information with too much granularity also leads to information overload. On the other hand, excessive data aggregation leads to the loss of valuable information. We collected developer's experiences and opinions and performed a qualitative analysis of the results to look for an ideal balance between providing information usable by developers to improve their test suite without information overload. This RQ aims to find which information originating from mutation testing results is considered the most valuable and useful by developers.

For RQ3, interpreting the results must be effective and efficient to seamlessly integrate MT into a developer's workflow. Presenting MT results effectively and efficiently enables a positive developer experience. We aim to collect developer experiences and opinions, perform a qualitative analysis on presenting MT results, and explore which aspects are important of the visual elements used to display information. This RQ aims to find the best way to convey MT information to developers by focusing on presenting complex information from MT.

Note that RQ2 focuses on what information should be conveyed, answering questions such as "Are mutation scores aggregated by mutation operators useful for developers?". In contrast, RQ3 focuses on how to present the information, answering questions such as "Is a line plot that shows the evolution of mutation scores easy to understand and utilize for developers?".

4.2 The Case

The case study is performed at the partner company Zenseact[1], which develops safety-critical software for the automotive industry. The company is owned by Volvo Cars and develops both Advanced Driver Assistance Systems (ADAS) and Autonomous Driving (AD) software products, mainly written in C++. When writing, the company had 580 employees, who mainly worked at an office in Gothenburg, Sweden [27]. Safety-critical development in the automotive industry necessitates adherence to various safety standards, such as ISO 26262 [28], as part of a comprehensive approach to ensuring user safety and the safety of the environment in which the software operates.

With a software stack containing many safety-critical components, validation is paramount to Zenseact's development process. As such, they pursue extensive testing of the software they develop to validate changes in the code, with manual reviewing, exploratory testing, and automatic testing in CI pipelines. Investing in various criteria for measuring test sufficiency is thus a high-value activity for them to determine whether code needs improved tests and as part of a larger argument for validation sufficiency. Due to this, some developers at the company are already familiar with MT, even though it has not been adopted company-wide.

[1] `https://zenseact.com/`

One aspect relevant to the study is that Zenseact utilizes *guardianship*, where a team is the main owner of a part of the code. This concept is also known as *code ownership* [29], where the owner is responsible for maintaining the code that it guards. During this study, guardianship was contextual information added to MT results to improve the developer experience.

The collection of data and in-depth analysis is done exclusively at said partner company. The conclusions offer an in-depth analysis and understanding of the integration and perception of MT results, specifically within the context of C++ safety-critical code in the automotive industry. The findings are designed to be applicable to similar settings. Table 4.1 provides an overview of the case study plan.

Table 4.1: Case study protocol.

Objective:	Facilitate the adoption of MT in industry settings
The case:	Integrating MT and presenting its results at Zenseact
Theory:	Mutation Testing Continuous Integration Data visualization
Research questions:	RQ1, RQ2, RQ3
Methods:	Observations (RQ1) Think-aloud sessions (RQ2, RQ3) Interviews (RQ2, RQ3)
Selection strategy:	Experiences from integrating MT at Zenseact (RQ1) Participating Zenseact Developers (RQ2, RQ3)

4.3 Research Method

In the following section and in Figure 4.1, the various activities and research methods are briefly summarized to give a general overview of the study. Afterward, each activity and method used is described in more detail.

(1) Integration of MT tools into the codebase at the partner company. We made observations during this process regarding the issues encountered and how they were handled. This step included integrating Mull, an MT tool, into the partner company's build system and developing code to parse, format, and upload results from MT to make them conveniently accessible from outside CI. We included the developed code in a CI pipeline that was executed daily. The MT results are presented using two tools: an HTML report generated by the MT tool and a Kibana dashboard developed during this phase.

(2) Analysis of observations generated in the previous step. We converted the observations into an experience report and created recommendations that address RQ1 based on the experience report.

(3) Interactive sessions in the form of think-aloud sessions with developers at Zenseact. Given the information and visual elements provided by the dashboard and report previously mentioned, the participants were tasked with assessing the quality of the test suite at Zenseact. While the participants carried out the tasks, they were repeatedly questioned about their thought process to collect data. The data consists of the developer's interactions and opinions on the dashboard and report.

(4) Interviews with the developers who participated in the think-aloud sessions. We used semi-structured interviews for the developers to reflect on their experience using the dashboard and report, focusing on the perceived benefits and issues.

(5) Thematic analysis of the interactive session and interview results. We then used the thematic analysis results to find patterns between the participants, which were grouped into themes and subthemes that address RQ2 and RQ3.

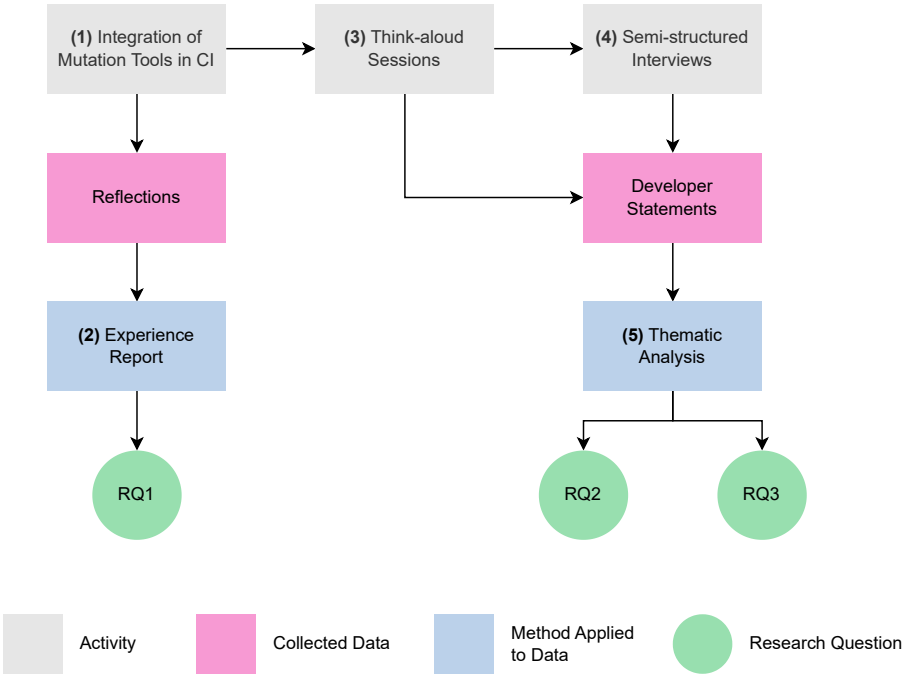


Figure 4.1: Overview of the work packages forming the case study.

4.3.1 Mutation Testing Integration (RQ1)

As mentioned, an experience report was created based on the observations made while integrating MT. The observations were made during various stages when adopting MT. The integration process was deemed finished once MT was integrated so developers could use the results to check for testing sufficiency without running MT themselves. This included merging all code required into the main Version Control System (VCS) branch and running the tool daily in CI on Zenseact's C++ codebase without errors related to the MT tool or code developed specifically for

MT. The code also had to be resilient to errors unrelated to MT occurring, such as the build system failing. Finally, once MT had run, the results had to be uploaded so developers could examine them.

Our observations focus on the challenges (or lack thereof) of the steps previously mentioned. A protocol was defined to collect the observations, which made the collection and format systematic. A set of events during which observations had to be made was also created to ensure that the observations remained on topic and that no valuable information was missed. The complete protocol is observable in Appendix B. Some key events documented during the observations were Decisions, Issues, Sentiments, and Discoveries.

To synthesize the experience report, the observations were sequentially analyzed and discussed with the partner company's developers for discussion, validation, and reflection. Together with their input, conclusions were formed and presented in the experience report.

4.3.1.1 Tools Used

Both Dextool[2] and Mull [3] were identified as viable options to use as MT tools, based on the previous research performed by Örgård et al. [19]. Early into the integration, various issues arose when using Dextool, even with simple projects. Due to the suitability for CI, ease of installment, features provided, maturity, computational cost, and compatibility with the Zenseact codebase, Mull was chosen as the primary tool to integrate at the partner company.

Two tools are used to visualize the results. The first tool is directly supported by Mull and consists of an HTML report. This can be considered the traditional way to present MT results and is covered in Section 2.3.2. The other tool used is Kibana [30], covered in Section 2.4. Kibana was chosen since it is already commonly used at the partner company, meaning that the developers have experience using the tool and would require less effort to become familiar with it. As a disadvantage, the types of visualizations used and experimented with are limited to those supported by Kibana. The data to be displayed was considerably hierarchical, something Kibana lacked visualizations for. For instance, nested treemaps and tables that allowed the exploration of hierarchical data were missing. Some simpler graphs, such as radar charts, were also unsupported. The said graph could have been implemented using Vega [31], a Kibana tool to create custom visualizations, but this was considered out of the scope of this research.

4.3.2 Evaluation of Mutation Testing Information and Visualization (RQ2, RQ3)

Think-aloud activities and semi-structured interviews with developers from the partner company were conducted to determine what information developers perceived as

[2] `https://github.com/joakim-brannstrom/dextool`
[3] `https://github.com/mull-project/mull`

useful from MT results and how it should be presented. Each participant first participated in a think-aloud and was interviewed after a short break. Together, they formed a single session, which took roughly 1 hour and 20 minutes per participant.

Before starting a session, the participants were asked for consent to record their voice and screen interactions. Furthermore, they were informed that they could opt out of the study by contacting the researchers at any time. Before publication, the draft was also sent to all participants so that they could provide feedback and corrections.

A session consisted of an initial introduction to MT so the participant was familiar with the essential concepts required for the subsequent activities. Some MT-related concepts (such as equivalent mutants) were not explained since they were not the focus of our research. Before continuing to the think-aloud session, the participants had the opportunity to ask for clarifications and to be asked if they fully understood all the concepts. This was done to guarantee they fully understood the concepts.

Then, the interactive think-aloud session was performed, where each participant interacted with the results from MT whilst sharing their thoughts. Finally, after a break, the interview was performed, where the participants shared their final reflections on the solution they interacted with in the think-aloud activity.

Only a subset of the mutation operators provided by Mull was used to carry out these activities. This was deemed sufficient since MT should be deployed incrementally as reflected in the experience report. Furthermore, three mutant operators were discarded since they caused Mull to crash unexpectedly during execution. The used mutation operators are listed in Appendix A.

4.3.2.1 Sampling

The following section follows the terminology defined by Baltes et al. [26]. The population of interest consists of software developers in the automotive industry. The sampling frame is the developers from the partner company, Zenseact. The sampling strategy followed was mainly convenience sampling, with elements of purposive sampling.

Convenience sampling, which consists of taking samples based on availability, was used to obtain samples for the sessions since the samples consisted of developers at our disposal; no developers from other companies were contacted or considered. However, elements of purposive sampling — sampling following a logic or strategy — were also used: a criterion was used to select the study participants to invite and improve the diversity of the samples. The criteria used consisted of the role, time at the partner company, seniority, and how often they interact with tests. This was used as the information and visualizations should not just be catered to those interested in MT but rather those who should be using it. This allowed evaluation from developers with different needs and perspectives. We expand upon this in Section 6.4.2.

The complete list of participants can be found in Table 4.2.

Table 4.2: Study participants.

ID	Role	Time at Zenseact	Overall User Experience	Test Interaction Frequency	Kibana Experience
1	AI support tool developer	1.5 years	3-4 years	Not often	Minimal
2	C++ computer vision engineer	7 years	7 years	Daily basis	None
3	C++ toolchain maintainer	3.5 years	12 years	Daily basis	Experienced
4	C++ and Java developer	7 years	8 years	Few times a week	Experienced
5	C++ developer and system architect	2 years	10 years	Few times a week	None
6	C++ feature developer	2.5 years	7 years	Daily basis	Experienced
7	C++ and Python developer	1 month	10 years	Weekly	None
8	C++ feature developer	1 year	6 years	Daily basis	Minimal
9	C++ feature developer	2.5 years	5 years	Weekly basis	Minimal
10	C++ feature developer	2.5 months	10 months	Not often	Minimal

4.3.2.2 Think-aloud Procedure

Before starting the interactive activity, we asked the participant some initial questions to learn more about their role at Zenseact and their professional experience. The participants were then informed that MT was run on the partner company's C++ codebase and were given a list of tasks where they had to interact with the results of the MT runs. During the tasks, the participants had to assess the test sufficiency of different teams, directories, and files. All participants were assigned the same team, directory, and file to make their experiences comparable. However, once the participants had finished their given tasks, they were free to look at the MT results of their own interest, such as those of their own team.

Before they started the tasks, they were given a short tour of the dashboard and the report. The tour aimed to demonstrate navigation through the dashboard and report without providing specific instructions on their usage.

While carrying out the tasks, they were instructed to actively share their thoughts, such as their impressions, what information they were interpreting, and what they thought of the visual aspects. If the participant didn't speak for an extended time, questions were asked to get them to speak again. For instance, "What are you trying to do right now?", "What are you looking for?" or "What are you thinking right now?". While carrying out the tasks, the participant had a summary of MT concepts and the list of tasks at their disposal and could also ask for clarification or guidance at any time.

To mitigate potential issues caused by the tools used to display results, the researchers defined a list of situations in which they would intervene. For instance, if the participant accidentally navigated out of the dashboard, we would re-open it. The full procedure for the think-aloud session can be found in Appendix C.

4.3.2.3 Interview Instrument

Before starting the interview, participants were again asked for consent to record their screen and audio. During the interview, the participant could still interact with the dashboard and report, making it easier to refer to specific elements.

Since RQ2 and RQ3 focus on information and visualizations, respectively, before any question was made, the distinction between the two was made. Furthermore, the participants were informed that they could ask for clarification on the questions they were given.

The semi-structured interviews allowed us to pursue additional information when the participants gave interesting answers for which we wanted additional reflection. The questions were divided into three main sections: the information presented, the visualizations used, and the final reflections. The questions mainly focused on the perceived usefulness and interpretability of the MT results. Once the interview had finished, the participants were given more details about the study and were informed about which parts of the solution they interacted with were developed by us.

The full interview instrument design can be found in Appendix D.

4.3.2.4 Think-aloud and Interview Reliability

Two different activities were carried out to validate the think-aloud session and interviews. Firstly, expert reviews were carried out by the academic supervisors, who are experienced in research and familiar with MT. Their feedback ensured that the data we obtained would help answer the research question, preventing bias originating from leading questions and avoiding vagueness.

Afterward, pilot studies were performed, during which both the think-aloud and interviews were carried out three times. Two of the pilot sessions were conducted with the study's industry supervisors, who were both familiar with MT. Their feedback focused on the structure of the think-aloud and interviews. Lastly, one trial run was conducted with a developer from the partner company unfamiliar with MT. Their feedback mainly concerned aspects they found vague or difficult to understand related to MT.

4.3.2.5 Reflexive Thematic Analysis

The thematic analysis was performed using reflexive thematic analysis, as described by Braun et al. [2]. Following their suggested process, the specific variation is described in Table 4.3.

Table 4.3: Variation used within reflexive thematic analysis as described by Braun et al. [2].

Aspect	Variation Used
Orientation to data	Inductive, the codes were created and refined during the coding process.
Focus on meaning	A semantic approach was initially and gradually moved towards a latent approach when a greater understanding was formed.
Qualitative framework	Experimental, exploring people's perspectives and what they believe.
Theoretical framework	Relativist end of the spectrum to investigate what people think and express, which might not reflect how they would actually use the results.

To aid the reflexive TA, a reflexive research journal was used to reflect on the coding process and examine assumptions held. The noted-down assumptions were: "people often don't want to learn new tools", "people might be more positive towards the developed solution as they try to be nice", "I'm a developer, so I assume people know about code", and "I assume they understood what mutation testing is".

Following the author's definitions, the following terminology is used:

Coding is the process of highlighting relevant parts of the transcriptions and assigning code labels to them.

Code labels are short identifiers attached to a highlighted part of the text. A code label always has a code.

Codes are in-depth descriptions of each code label.

The voice recordings were transcribed using Whisper[4], a speech recognition AI. Each automatically transcribed text document was manually checked to correct transcription errors. The screen recordings were used to clarify what participants referred to in the voice recordings.

Once the transcriptions were finalized, Taguette [32] was used for the coding process, a free and open-source tool for qualitative analysis research. The inter-rater reliability was evaluated before starting the coding process. Multiple passes were performed with multiple discussions and iterations of the codes and code labels.

Once the coding process was deemed sufficient, we created a thematic map using Miro[5] (a collaborative digital whiteboard tool) to group the codes into themes. All

[4] `https://openai.com/research/whisper`
[5] `https://miro.com/`

highlights were exported to sticky notes along with their corresponding code labels, and each note was colored by the interviewee ID. The color indicated how a code was distributed; is it a consensus or an outlier? Clear outliers were disregarded. The themes were created and iterated upon in multiple phases until a final set was created and presented in the results.

Inter-rater reliability had to be ensured, meaning that both coders had to agree on which parts of the transcriptions were relevant and agree about the code labels given to the highlighted parts. For the agreement on relevant parts of the transcriptions, Krippendorff's Alpha (α) [33] was employed, which allows measuring the amount of agreement between coders. Krippendorff's Alpha is a flexible metric, as it allows a variable amount of coders, different types of input data (nominal, ordinal...), missing values, and considers the agreement that might happen by chance instead of actual agreement.

To obtain the measurement, both researchers independently coded the same session. A table was created where each column pertained to a researcher, and each row represented a highlight that either a single or both researchers highlighted. When only one researcher highlighted the text, the values differed. Encoding the results using this format allowed us to compute the measurement.

When Krippendorff's Alpha was computed, it resulted in $\alpha = 0.45$, with a 95% confidence interval of $[0.25, 0.61]$. This was deemed insufficient, following the guidelines for interpreting the value suggested by Krippendorff [33]. We then discussed the disagreements that we had to reach a common ground. Afterward, both coders independently coded the second session, and Krippendorff's Alpha was again computed, which resulted in $\alpha = 0.72$, with a 95% confidence interval of $[0.54, 0.86]$, which was considered sufficient.

For the agreement of the code label assigned to each highlight, discussions were carried out when different code labels were assigned. Most conflicts were related to different formatting or code labels, which meant the same thing. Discussions were held about disagreements related to the meaning of the text until a common understanding of the text was reached. Since each coder passed each transcription at least once, they also checked the labels assigned to highlights to look for disagreements, thus improving and validating the other's annotations.

5

Results

This chapter covers the study's results. First, we present the solution developed to integrate MT into CI and the Kibana dashboard that displays the results. Afterward, we present the experience report related to RQ1. Finally, we present our findings from the think-aloud sessions and interviews that we performed related to RQ2, RQ3.

5.1 Tool Integration into CI Pipeline

The following section describes the sequence of operations executed in the created CI pipeline. Refer to Section 5.3 for the experience report that reflects on the pipeline creation described here and the rationale behind the design decisions.

In this case study, MT is applied in a nightly build. Periodic MT was chosen due to contextual factors: the codebase includes safety-critical code, which must comply with safety standards that impose a minimum degree of testing. Periodic MT runs all test targets periodically during times when most CI resources are idle, which is less computationally heavy than commit-aware MT. With commit-aware MT, tests must be run each time a commit is pushed to the remote VCS.

The sequence in the pipeline, observable in Figure 5.1 is as follows:

(1) Find all C++ testing targets buildable in Zenseact's codebase using Bazel, the build system used at the partner company. Each target is individually compiled using Clang to convert the C++ code to LLVM IR. This produces a separate executable for each target. Mull's default behavior causes it to mutate the code under test and its dependencies, something described in more detail later on in Section 5.3.2.2. The tool's behavior was changed so as not to mutate the dependencies. For each test executable, the tests are run per mutant, and a report is generated. A file, `mull_metadata.json`, references all reports generated with some additional metadata, such as a timestamp.

(2) As every testing executable generates its separate report, the reports are merged into a single file `merged_report.json`. This file is then displayable using the HTML report. This report was one of the two ways the host company developers had to interact with the results, the other being a Kibana

dashboard.

(3) The merged report is transformed into a more suitable format to be uploaded to Kibana. The report is also enhanced with additional information that enriches the MT results during this step. For instance, each surviving mutant has its guardian team associated with it. Analyzing the report with this information allows teams to find mutants related to the code they are responsible for.

(4) Upload the analyzed report to Kibana, allowing access to the results.

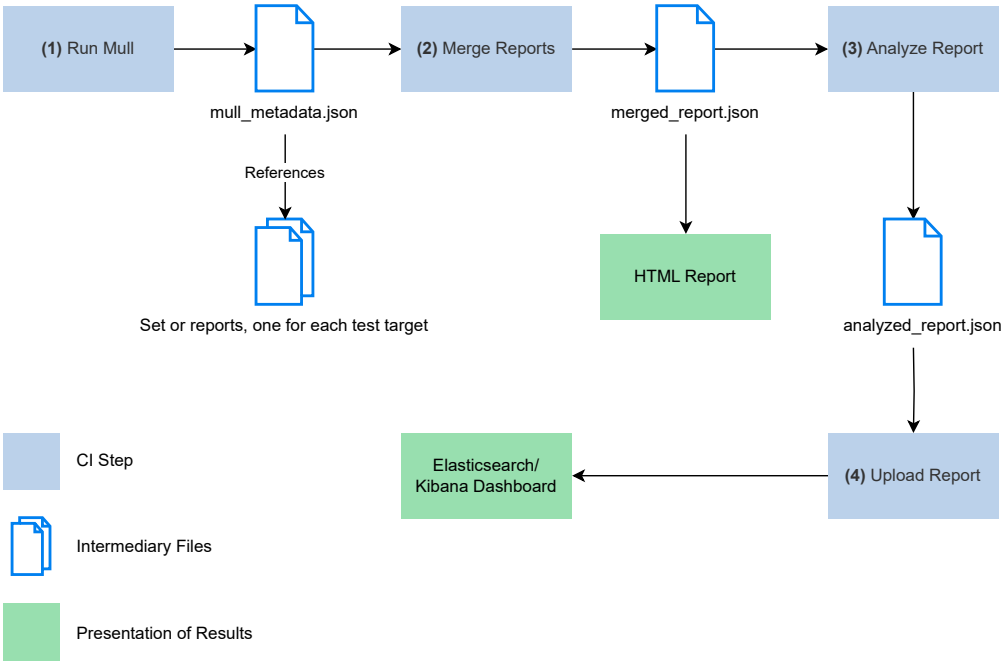


Figure 5.1: Sequence of operations in the CI pipeline and by-products.

In summary, the pipeline runs MT on the host company's C++ codebase. The results are combined into a format compatible with Mutation Testing Elements. Then, they are analyzed and reformatted into a format suitable for Kibana. Finally, they are uploaded to Kibana, all without human intervention.

5.2 Developed Kibana Dashboards

RQ2, RQ3 aim to evaluate what information from MT results to display and how they should be visualized. The following section presents the data visualizations created in a Kibana dashboard and the information included within them. It should be noted that the information from the HTML report is also evaluated for RQ2, RQ3, although the report was not created during this project. The information and visualizations used for the HTML report can be found in Section 2.3.2.

The developed dashboards are interactive, allowing users to navigate between them and filter data. This allows the scope to be narrowed to an area of interest/level of

detail, such as a team, directory, or file. This scoping can be performed by either selecting a certain value, such as a team, file, or mutation operator. This will narrow the scope in Kibana to values matching said filter. Furthermore, there are dropdown lists that allow users to filter in a similar manner, which is used to, for example, select a team.

As terms such as information and widgets can be ambiguous, the following definitions are used:

Information: A certain value/metric. An example is the mutation score or lines of code for a file. Said value can be visualized in multiple ways, such as with a bar or line chart.

Data visualization: The generic technique used to display certain data. For example, a line chart visualizes a value over time.

Widget: This refers to the specific combination of information displayed using a specific data visualization technique. For example, a line chart can be used to visualize the mutation score of a file over time.

5.2.1 Information, Data Visualizations and Widgets

The information items in Table 5.1 are gathered per run and can thus be tracked over time. As previously explained, we utilize guardianship. This means that all the information below is connected to a team, allowing it to be scoped. Furthermore, everything below has a path, allowing the data to be filtered, which is utilized in the developed dashboard.

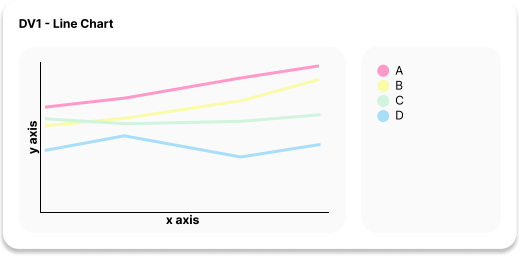
Table 5.1: Information from MT.

Category	**Information**
Metadata	Date when the analysis was run
	The commit of when the analysis was run
Mutation Score	Mutation score
Mutant	Whether it survived or was killed
	Path to the file
	Location in file (row and column)
	Mutation operator
	Replacement value
	Number of surviving mutants

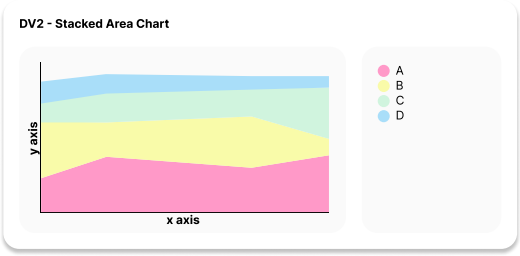
The used data visualization techniques are presented in Table 5.2, with examples.

Table 5.2: Data visualizations.

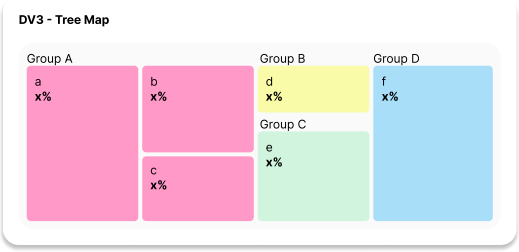
Visualization	Explanation	Example
Line chart	Used for visualizing values over time.	Figure 5.2a
Stacked area chart	Used for visualizing values over time that, in addition to the line chart, also display proportions.	Figure 5.2b
Tree map	Used for visualizing areas of interest with proportional boxes. This gives an overview of multiple areas, which can aid in spotting the "greatest" area in some context.	Figure 5.2c
Table	Used for visualizing grouped data.	Figure 5.2d
Pie chart	Used for visualizing proportions.	Figure 5.3a
Single value	Used for visualizing a specific single value, for example, a date.	Figure 5.3b
Gauge	Used for visualizing a value between a range. Similar to the single value visualization, it gives a ratio of where the value is within an upper and lower bound.	Figure 5.3c
Dropdown	Arguably not a visualization but rather an input. Is used for displaying a set of options and allows a selection.	Figure 5.3d



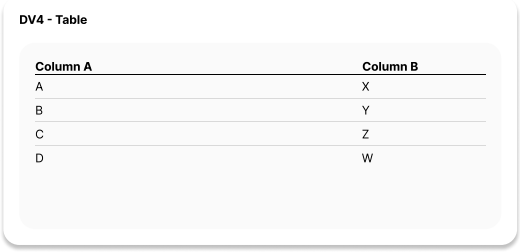
(a) Line chart.



(b) Stacked area chart.

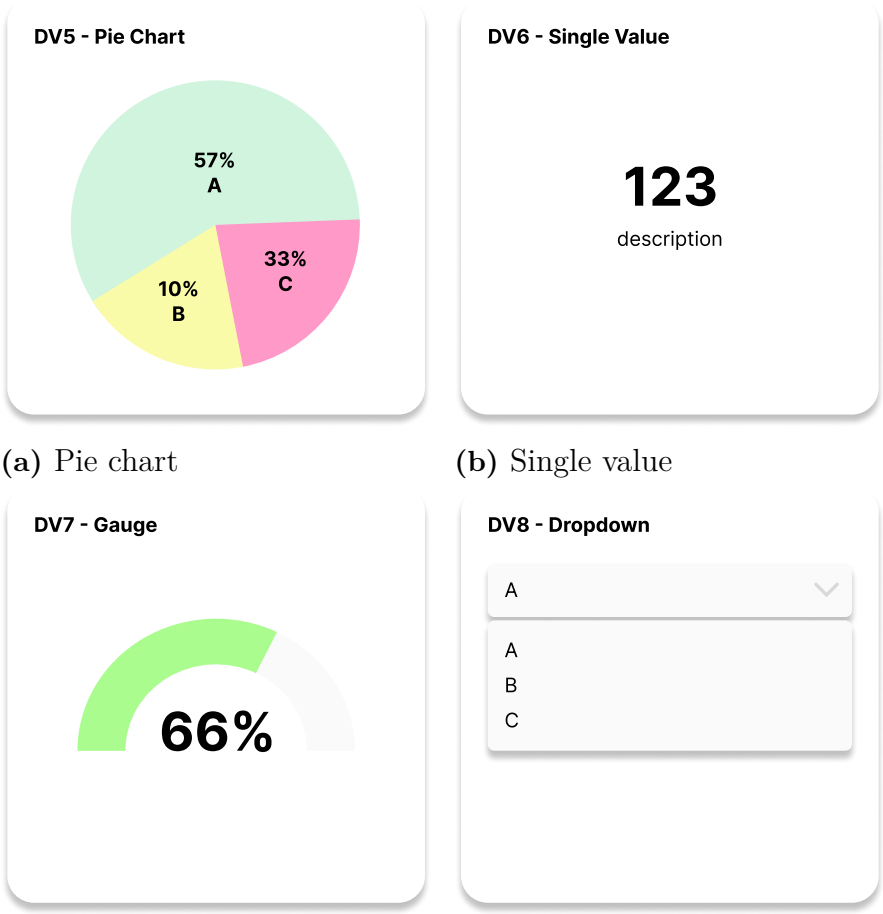


(c) Tree map.



(d) Table.

Figure 5.2: Examples of a line chart, stacked area chart, tree map, and a table.



(a) Pie chart

(b) Single value

(c) Gauge.

(d) Dropdown.

Figure 5.3: Examples of a pie chart, single value, gauge, and a dropdown.

The specific widgets used within the dashboards are presented in Table 5.3, including the data visualization technique and what information is provided.

Table 5.3: Widgets.

ID	Name	Visualization	Description
		Team and Directory Dashboard	
W1	Filter	Dropdown	Filter values by team (in the team dashboard) or file (in the directory dashboard).
W2	Mutation Score	Gauge	The mutation score. Aggregated by team, directory, or file depending on filters.
W3	Date	Single value	Date of the results.
W4	Commit	Single value	The specific commit the analysis was run on.
		Team Dashboard	
W5	Overall Team Mutation Score	Line chart	Mutation score per team over time.
W6	Mutation Score per Directory	Line chart	Directories and their mutation score over time.
W7	Surviving Mutants per Directory	Stacked area chart	Number of surviving mutants per directory over time.
W8	Mutation Score per Directory	Line chart	Directories and their mutation score.
W9	Surviving Mutation Operators	Stacked area chart	Number of mutation operators over time.
W10	Number of Surviving Mutants	Single value	Number of surviving mutants.
W11	Proportions of Surviving Mutators	Pie chart	The proportions of surviving mutation operators.
W12	Directories and Files with the Most Surviving Mutants	Tree map	Hierarchical view of files with the most surviving mutants.
W13	Surviving Mutants	Table	The surviving mutants, including their location and mutation operator.
		Directory Dashboard	
W14	Overall Directory Mutation Score	Line chart	Mutation score scoped by a directory over time.
W15	Mutation Score per File	Line chart	Mutation score per file over time for a directory.
W16	Surviving Mutants per File	Stacked area chart	Number of surviving mutants per directory over time.
W17	Surviving Mutation Operators	Stacked area chart	Number of mutation operators over time for a directory.
W18	Mutation Score per File	Table	Mutation score per file for a directory.
W19	Number of Surviving Mutants	Single value	Number of surviving mutants for a directory.
W20	Proportions of Surviving Mutation Operators	Pie chart	The proportions of surviving mutation operators for a directory.
W21	Files and Operators with the Most Surviving Mutants	Tree map	Hierarchical view of files with the most surviving mutants for a directory.
W22	Surviving Mutants	Table	The surviving mutants, including their location and mutation operator for a directory.

5.2.2 Dashboards

Two dashboards were created in Kibana to visualize the MT results, one for teams and one for directories. The team dashboard was used to narrow down the scope of the vast number of mutants generated from running Mull on the codebase to display only relevant information to a developer. The team dashboard is presented in Figure 5.4.

The directory dashboard is one granular step below the team dashboard. This dashboard was developed to show more specific MT data for a directory to aid

the process of evaluating the test quality of a specific part of the code and aid in improving the test quality. More granular data, such as mutation score per file, is displayed, and data is limited to the specified directory. The dashboard can be seen in Figure 5.5.

Developers can navigate from the team dashboard to the directory dashboard. To do so, users can use widgets in the directory dashboard. As such, a developer can first see the mutation score of all their team's directories in the team dashboard and then navigate to the directory dashboard for a directory of interest.

It should be noted that the dashboards presented in this study are not the Kibana dashboards used by the participants during the study, but instead illustrations used to exemplify what was created. Furthermore, the line plots and stacked area charts shown do not include any values (dates, mutation scores...) on either axis for the sake of simplicity.

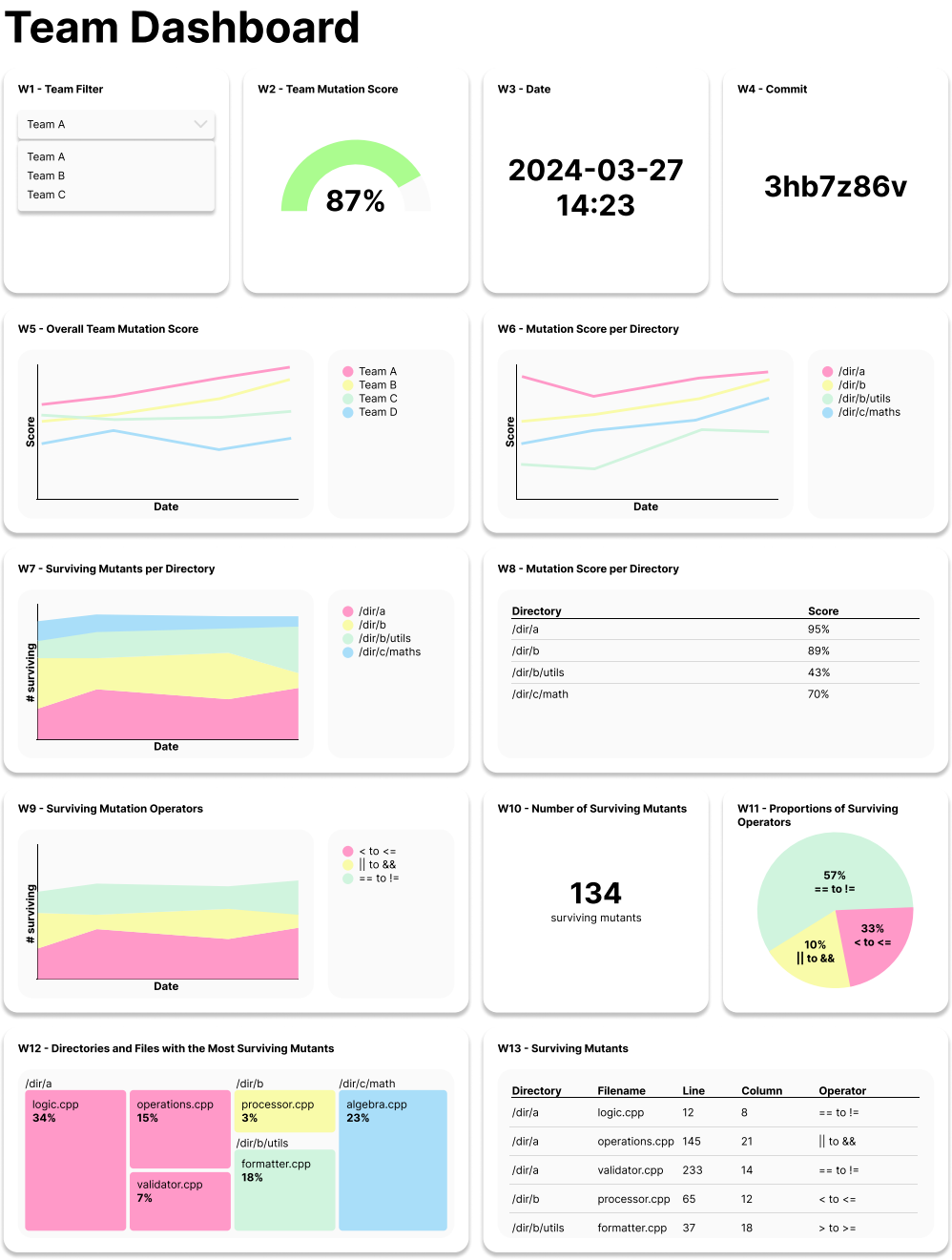


Figure 5.4: Team dashboard with synthetic data.

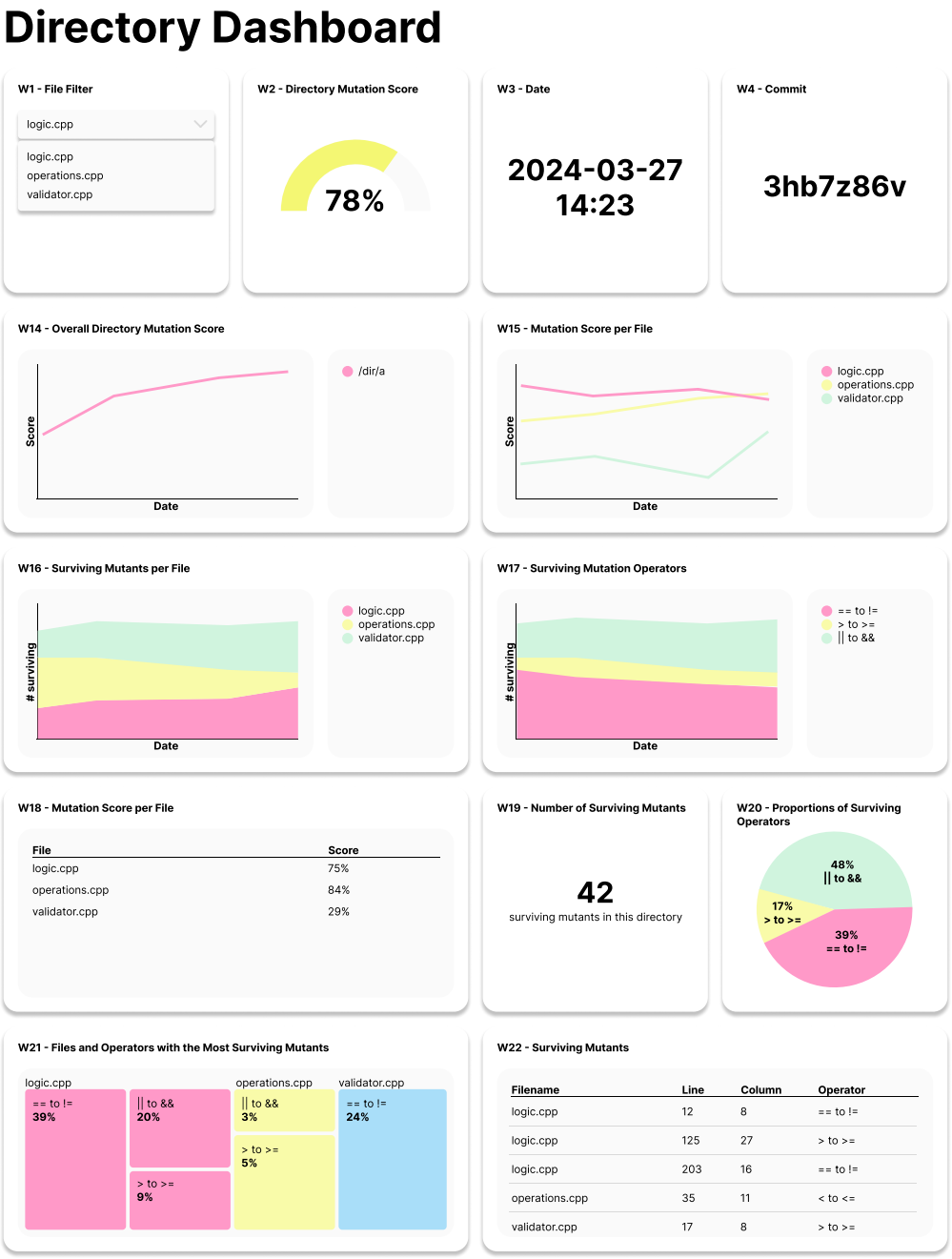


Figure 5.5: Directory dashboard with synthetic data.

In summary, two different but related Kibana dashboards were developed with interactive elements such as filtering and navigability. The dashboard contains visualizations that display information on MT results and was designed to provide an overview of the MT results without overwhelming the developer.

5.3 Experience Report: Integrating Mutation Testing Tools Into Build Systems (RQ1)

The following section covers the experience report related to RQ1, which reflects on the pipeline creation described in Section 5.1, including integrating MT into a codebase and a CI pipeline. We made 34 observations during the engineering process, which were used to create the report. We, the researchers, primarily made the observations, with some contributions made by two developers from the host company.

5.3.1 Project Kickoff

We began by evaluating the MT tools Dextool and Mull, which had shown promising results in previous work [19]. We considered both tools as starting points and created a minimal proof-of-concept for each one to experiment with using simple C++ code. Mull was used with minimal issues.

On the other hand, we had problems getting Dextool running, possibly caused by a mismatch between the bindings and the library. Running Dextool started the MT process but failed with an error message of `error: exiting...` without stating the error, making it hard to debug and progress. The exit left the files in an intermediary step with flags to disable or enable specific mutations, as shown below in Listing 5.1. This was likely a user error during the installation or with clashing dependencies. However, resolving the issue without a helpful error message was hard. The installation was also attempted on multiple Linux distributions: EndeavourOS (Arch), Pop!_OS (Ubuntu-based), and Ubuntu.

```
static bool greater_or_equal(const double a, const double b) {
    if (unlikely(dextool_get_mutid() == 1250712460u)) {
        return true;
    } else {
        return a >= b;
    }
}
```

Listing 5.1: Example of a Dextool flag left when exiting.

We then started attempting to apply the MT tools to the partner company's codebase, where we tried to get Mull to work on a subset of the source code as a proof of concept. Again, the usage was straightforward, and we could apply MT using Mull on the first day. We also attempted to use Dextool despite the issues we had previously encountered, but we again encountered the same problem that we could not fix. Due to the blocking factor and the fact that Mull was already showing signs that it could be an appropriate tool, we decided to focus only on the integration of Mull.

One observation we made was that to get Mull working, most of the knowledge required was related to the build system, not the tool itself. The tool's installation

guide mentioned any information needed regarding the tool itself. As such, the biggest challenge related to making the build system use Mull was due to the build system itself instead of Mull, which is expanded upon further in Section 5.3.4. We therefore recommend that the integration should be done by developers familiar with the build system rather than MT experts, as we found that minimal knowledge of MT is required to integrate the tools.

5.3.2 Friction Between the Build System and Tool

Due to Mull's initial success, the next step was applying MT to the entire codebase instead of just one testing target. To detect and solve issues early, we wanted to follow an incremental strategy by starting with a very small subset of mutation operators and testing targets and expanding gradually. This strategy reduced the feedback loop and the computational cost during the integration process, enabling us to see more quickly whether our code could successfully run from start to finish.

Due to this strategy, we detected a minor number of specific test targets early on, which caused Mull to crash. This was because of technical reasons, such as the use of hardware-specific code, which we then excluded by specifying a list of targets to exclude. Mull also had problematic behavior when applied to CUDA code, which is used to program using NVIDIA GPUs.

However, a disadvantage of using this strategy was that we were unaware of how long it would take to perform MT on the entire codebase initially. Another concern related to manual exclusion was that it is not a scalable solution: new testing targets could appear at any time, which causes the MT tool to crash. As such, a more scalable solution was implemented that dealt with the tool's crash for specific targets by programmatically detecting errors in the generated logs.

We found all test targets using a Python script by querying the build system and then applying MT to each test target found. However, we encountered two issues related to Mull that had to be addressed.

5.3.2.1 Multiple Reports, One Result

Mull can only apply MT to a single test target and generate a corresponding report instead of applying MT to multiple test targets and generating a single report for all of them. Since our goal is to provide useful and convenient results to developers, we had to address this by merging multiple reports into a single one. To do so, we still run Mull on every test target individually and then use a Python script we developed to merge the reports into a single, centralized report, still following the Mutation Testing Elements schema format.

This issue was considered an unexpected limitation to Mull. It is unknown why Mull doesn't support generating a report out of multiple testing binaries. Still, it negatively impacts the tool's usability by making it more difficult to use in certain workflows.

5.3.2.2 Redundant Mutants

While developing the merging Python script, we noticed that the same mutant[1] was run in different testing executables. After some investigating, we discovered that Mull mutates source code directly invoked by the testing executable as well as its dependencies. This is demonstrated in Figure 5.6: although the purpose of `test_lane_assist.cpp` is to validate `lane_assist.cpp`, Mull will still mutate `sensor.cpp` too during compilation. This leads to additional computational costs and a dilemma: should the mutant be considered killed or surviving if a mutant survives in one test executable but not in another?

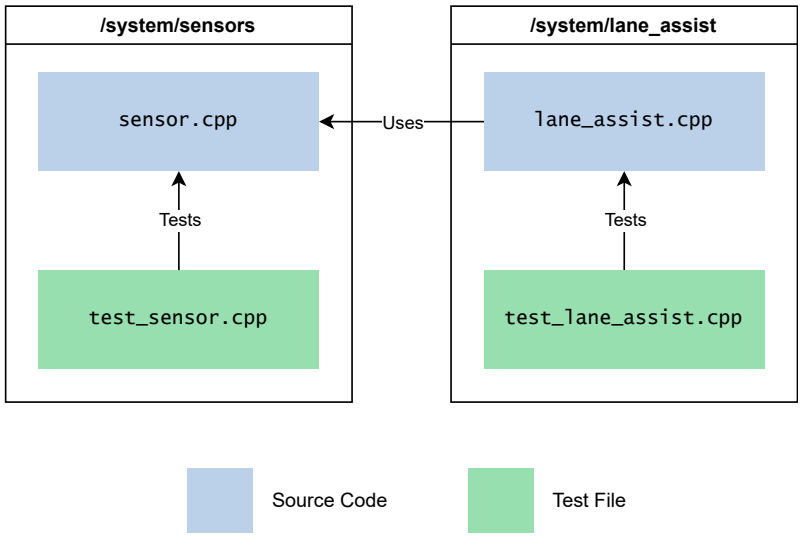


Figure 5.6: Example scenario where mutations are tested multiple times with default Mull settings.

One possible solution would have been to accept this behavior and implement mechanisms in the merger script to deal with this situation. For example, if the same mutant appears in different reports and is killed in even a single one, it is considered killed. Another solution is to only mutate direct dependencies, also reducing the number of mutants and thus the computational cost.

We chose the latter approach using one of Mull's features to address this: whitelisting. A new whitelist is generated for each test target compiled, making Mull only mutate the source code on which the testing target directly depends. This prevented mutants from appearing multiple times, making the MT pipeline run roughly 40% faster. However, this required more engineering effort to automatically update the whitelist for each testing target by detecting which source code it tests.

[1] We consider "same mutant" to be a mutant reported in the same file, line, column, mutation operator, and replacement value.

5.3.3 Presentation of Results

We chose to use Kibana as the visualization tool to create the dashboard. Kibana was chosen since developers at the partner company were already familiar with it, and used to track other aspects of the codebase, such as technical debt. Extra steps were required to get the information from the merged report into Kibana, as the report format was unsuitable for directly uploading to Kibana. Furthermore, changing the format was also advantageous: the report follows a format specifically designed for MT results, which we wanted to enhance with additional information incompatible with the format. We accomplished this by breaking away from said format to one more suitable to Kibana. By enhancing the MT results with information such as the guardian of each source code file, the dashboard presenting the information became more interactive and dynamic as developers could filter the information. For instance, they could filter to only see the MT results of code maintained by a specific guardian.

Another observation was that the Kibana dashboard was independent of the MT tool. Given a generic frontend to present results, the backend MT tool becomes interchangeable. As such, choosing the MT tool to use becomes more flexible. However, this does imply a conversion from the format of the MT tool to the format usable to the frontend.

5.3.4 Integration Into CI

We attempted to run the developed pipeline in CI only after its development was complete. This led to issues as we encountered CI pipeline divergence: we had different behavior when running locally and in the CI pipeline. We then had to investigate and solve the issue, which took significant time.

We attribute this to us following different strategies for the development and integration (into CI) parts. During development, an incremental strategy was followed while, on the other hand, everything was integrated at the same time into CI. Finding the cause of issues became challenging, as it could be anywhere in the developed pipeline. Furthermore, there was a long feedback loop since commits had to be merged into the main branch before they would affect CI pipelines.

The issue was caused by a combination of details related to both the build system (Bazel) and Mull:

- Mull performs MT in two separate steps: compilation and execution. In the compilation, Mull also references the source code location of each mutant generated. This includes the file path, line, and column. The second step executes the tests and checks whether each mutant survives. For each mutant, it also performs a sanity check, checking that the source code file of the mutant exists and that the line for the mutant exists.

- Bazel is a build system where a target, its dependencies, and the steps to obtain the target are defined. To build a target, Bazel moves all the defined

dependencies into a sandbox to guarantee that no implicit dependencies could accidentally be used.

When building locally using Bazel, the filesystem inside the sandbox remains unchanged. However, in CI, the filesystem inside the sandbox is different. In CI, targets were compiled inside the sandbox and then executed outside the sandbox. Since the target was built inside the sandbox, it saved references to source code files using file paths that were no longer valid. Mull attempted to open source code files using invalid file paths, which caused it to fail. To solve this issue, both the compilation and execution of test targets were performed inside the sandbox.

While fixing the CI pipeline, we manually ran our developed code locally daily to update Kibana's results. This was necessary as we required data to be used during the think-aloud sessions for RQ2, RQ3. Having to run the code manually highlighted the value of CI in this context to us; even though starting the MT process was effortless, it was still cumbersome due to daily repetition and the time of running MT.

5.3.5 Retrospective

Most of the time dedicated to integrating Mull into the CI pipeline was related to writing Python scripts to circumvent unexpected issues caused by the unique combination of codebase, build system, and MT tool. Although we got Mull working on the code on the first day, all of the issues encountered were related to making the tool behave as intended. Consequently, much work had to be dedicated to successfully running the MT tool and handling the results, significantly more than integrating a code coverage tool. Although Mull was considered suitable for CI, we do not consider the tool mature enough to be easily inserted into a complex codebase with little effort. This is something we deem problematic: MT tools are not a central tool of a codebase, and developers should not have to spend excessive time to get them working.

We also reflected on how feasible it would be for the integration to keep working after a period of time. The main reason the integration could break would be upgrading the version of any of the tools used at Zenseact, such as Bazel, Clang, and LLVM. Mutation testing will likely (and understandably) not be considered when said upgrade is performed. For the integration to be fixed, it will need to provide value. Without value, no developer would notice that it broke and would inevitably be removed. On the other hand, if the integration is actively used and provides value, people will investigate it when it breaks since they are missing valuable information from their workflow. As such, we conclude that for MT to survive in the long run, the initial integration should focus on providing value, as without it, it is doomed to break and be forgotten.

5.4 Think-aloud and Interviews Thematic Analysis (RQ2, RQ3)

In the following section we present the themes and subthemes that emerged from thematic analysis, given the results from ten think-aloud and interview sessions. The objective of the think-aloud and interview sessions was to discover what information developers consider valuable when using MT and how it should be presented. We also focused on why they considered the given information or visualizations valuable. An overview of all themes, subthemes, and how they are related to each other can be found in Figure 5.7, while the description of the final themes can be observed in Table 5.4 and the description of the subthemes can be found in Table 5.5. It should be noted that *the solution* refers to the combination of the previously described HTML report and the Kibana dashboard.

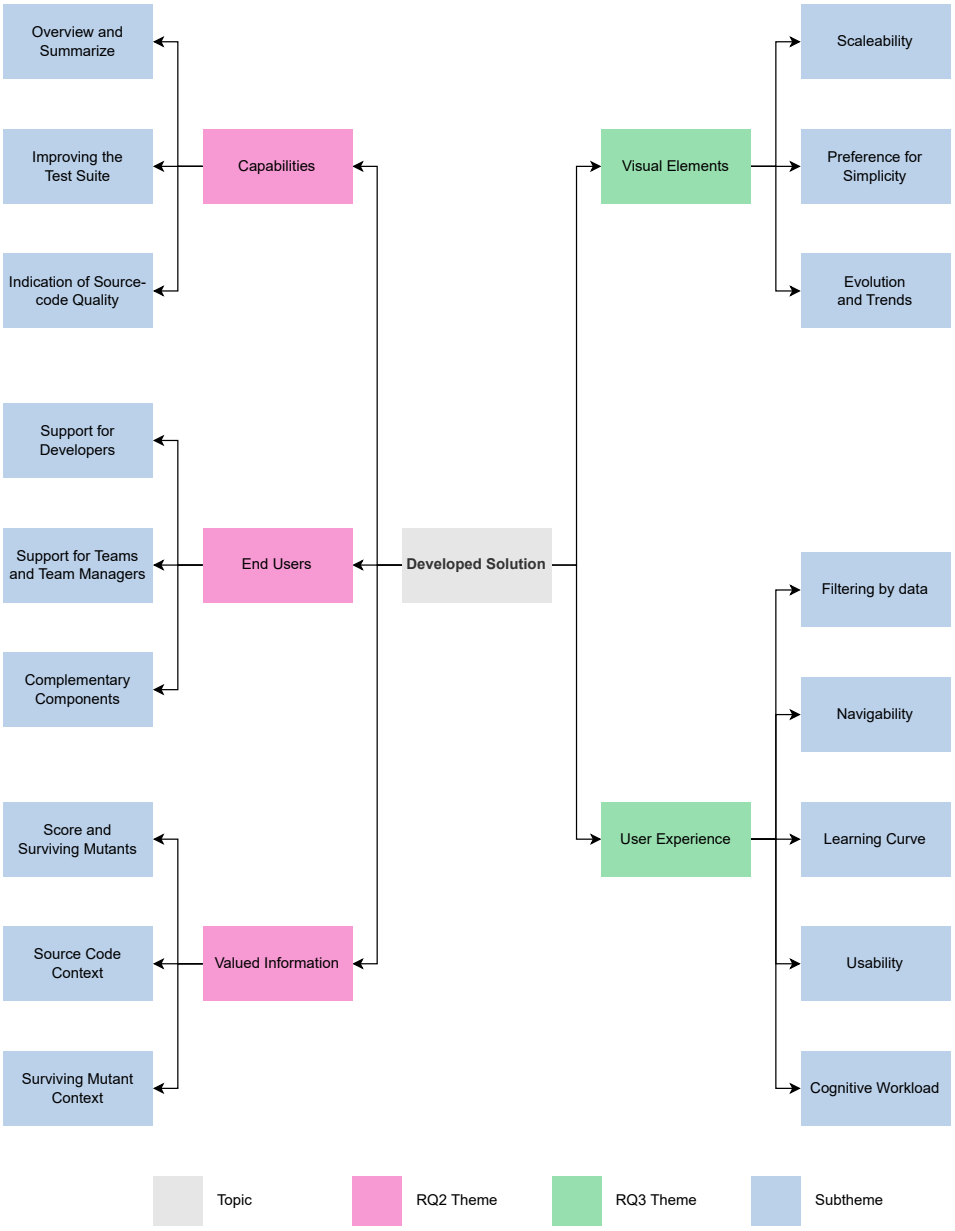


Figure 5.7: Overview of identified themes and subthemes.

Table 5.4: Description of themes.

Theme	Description
Capabilities	What the provided information of the solution enables users to do
End Users	To whom in a software development environment could the solution provide value to
Valued Information	What information from the solution was perceived as useful, and what was missing
Visual Elements	Which factors were important when visualizing information
User experience	Which factors were important when using the solution

Table 5.5: Description of subthemes.

Theme	Subtheme	Description
Capabilities	Overview and Summarize	Ability to see the overall status of the test suite
	Improving Test Suites	Ability to improve the quality of the test suite
	Indication of Source Code Quality	Ability to estimate the quality of source code
End Users	Support for Teams and Team Managers	Value of MT for teams and team managers
	Support for Developers	Value of MT for developers
	Complementary Components	The relation between the report and dashboard
Valued Information	Score and Surviving Mutants	Perceived usefulness of mutation score and specific surviving mutants
	Source Code Context	The want of more context related to the source code
	Surviving Mutant Context	The want of more context regarding the surviving mutants
Visual Elements	Scalability	How developers perceived the scalability of the solution
	Preference for Simplicity	Preference of the developers regarding visualization complexity
	Evolution and Trends	Thoughts on showing MT results over time
User Experience	Filtering by Data	Ability to filter information in the dashboard
	Navigability	Interconnectedness of the solution
	Learning Curve	The learning curve associated with both the solution and MT
	Usability	Ease of use of the solution
	Cognitive Workload	Mental effort required to use the solution

5.4.1 Capabilities

This section discusses the ways in which the solution is considered beneficial.

Overview and Summarize In software development environments, developers aim to keep their test suite up to a certain quality standard that they consider sufficient. With modifications in the source code and tests, tracking whether all tests are still sufficient can become challenging. Participants found the dashboard

particularly useful for getting an overview of the entire codebase, seeing in which direction the quality of the tests is evolving, and it allowed them to locate regions that needed improved testing the most.

> With the dashboard, it was quite easy to see how it's improving over time.
>
> - Participant 1

> I would probably monitor the dashboard more often, sometimes, just to see trends and changes and stuff.
>
> - Participant 5

Improving the Test Suite Eight developers stated that the provided report enabled them to improve the tests, given the surviving mutants shown. They mainly argue in favor of this since the report allows them to see the specific surviving mutants in the source code, giving them sufficient guidance to make or modify tests.

> The report I would use to address, to improve the code that has been flagged, to have less survived and more killed. So increase the ratio.
>
> - Participant 3

> ... but [I prefer] the other [report] when working on improvements. Like now we will make sure to get 100% I guess I would use this one to check where we have these missing tests in detail.
>
> - Participant 2

> I think the use case for the report is that you will get very detailed information on which mutants that were not killed. And that is what I'm interested in, since I would like to improve the tests so that the mutations get killed, if possible.
>
> - Participant 7

Indication of Source Code Quality Although mentioned by fewer developers, another functionality brought up by the more experienced developers was that MT gave an indication of the source code quality. They speculated that regions or specific lines of code with dense mutants (whether surviving or not) are code smells and indicated technical debt. They argued that with dense regions of mutants, they could identify problem areas in the code, such as complex expressions in if statements, which, when refactored, would improve both readability and 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. So, it could help with refactoring the source code.
>
> - Participant 3

> So, since this is like a multi-multi-Boolean expression, I can suspect that you've tried some conditions, but it's harder to test all combinations [in this code]. ... Yeah, that's good information.
>
> - Participant 6

> I would probably also look into this file if I can already, maybe just reduce the mutation surviving by refactoring as well.
>
> - Participant 9

5.4.2 End Users

The following section reports on which professionals in a software development environment could use the solution to aid them in their tasks.

Supports Teams and Team Managers Nine developers thought the dashboard could be useful for more than just developers. Repeatedly, they mentioned that no in-depth knowledge of the code was required to interpret the dashboard. Consequently, it could benefit individuals in more managerial positions, such as Product Owners (PO) and team managers, to get insights into the state of testing over the teams they oversaw. Furthermore, they considered that teams could use the dashboard during standups to plan the next tasks they would focus on.

> I think the left one [dashboard] is also a bit more a PO thing, because they're more interested in status.
>
> - Participant 2

> But as a team, it would be good to look at the dashboards and assess if we need to prioritize working on this.
>
> - Participant 4

> To give an overview for people that does not work with the specific functions, maybe the dashboard is better, I guess.
>
> - Participant 7

Supports Developers Regarding developers, they considered that they would likely use the report more often than the dashboard. Their main focus was on the code and the tests they were responsible for rather than getting an overview. They also considered that specific knowledge about the code is required to use the report effectively.

> I would probably mainly be using the report, actually, because I would be interested in my own code, and I know where that is, ... because I know where to look.
>
> - Participant 4

> Probably the developer knows which components they guard and so on, so it's easy to navigate into that [report] and then, I mean, here you get most of what you need ...
>
> - Participant 6

Complementary Components Although most developers preferred the report, many also pointed out that they consider the dashboard and report to be complementary. They consider the dashboard an entry point where they can easily locate weak points in their test suite. Afterward, they can go to the report to observe specific surviving mutants in the source code and address them by modifying the test suite.

> I think I would be able to do everything with only the report here. 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.
>
> - Participant 1

> Like, you'd use them at different times and sometimes use both.
>
> - Participant 5

> So I would look at the dashboard to get an overview there, but then I would probably use this interface [report] to actually look at where the code ...
>
> - Participant 8

5.4.3 Valued Information

This section covers what information developers valued the most from the developed solution and what they felt should have been added to make the solution more valuable.

Score and Surviving Mutants The majority of developers indicated that they mainly used the mutation score and surviving mutants in the source code to assess

the quality of their test suite with the information available from the solution. Less focus was given to the number of surviving mutants, the mutation operators used, or metadata that enables replicability, such as the commit used to generate the MT results.

> And then yeah, it's, of course, interesting to see which kind of mutant survived.
>
> - Participant 9

> Mutation score is, for me, it feels like the most important.
>
> - Participant 10

Source Code Context Nine out of ten participants frequently mentioned additional context unrelated to MT when asked which additional information could have assisted them in their tasks. This included relating MT results to the results of Code Coverage (CC), the size of source code files, the criticality of the code, the complexity of the code, and the coding safety standard the code had to adhere to.

> Five is considered quite a lot here, but I mean, it depends how critical this is.
>
> - Participant 1

> If you could somehow include the GCOV report into this, then you could have like all testing in one place which would be nice maybe. At least double check that there is a test that covers it or not. ... Then you could quickly know if the mutant survives because I didn't have a test or my test was too bad. But maybe that could help me investigate more quickly "what should I do?"
>
> - Participant 6

> Maybe I've missed it, but like some other measurements to show more context in the number of lines or something. ... If you have very little code it's very like it's very easy and probably a little effort to fix ... while if you have a team that has like million lines of code then the amount of work to improve is quite significant.
>
> - Participant 8

Surviving Mutant Context Participants also mentioned that they wished for more information on each surviving mutant, mainly to prioritize which to address first. They wanted to quickly know how problematic each surviving mutant was and how much effort it would take to fix it. With the current information, they showed uncertainty with regard to what the best course of action would be.

> So, if you want to improve your old code you would want the tool to show: here are the ones that you should focus on first. ... You want to remove the most dangerous ones first and maybe eventually go down to zero.
>
> - Participant 5

> It is quite interesting to me to see like "How much effort I would need to put in a specific directory" for example, to add coverage.
>
> - Participant 8

> It's easy for us to get better percentage here, I guess, but I'm not sure.
>
> - Participant 7

5.4.4 Visual Elements

This section reports on which visual aspects were important for developers when interpreting MT results from the solution.

Scalability Eight developers reflected that the dashboard's visualizations still fulfilled their purpose despite the large scale of the partner company's codebase. They mainly attribute this to the filtering functionality, which allows them to see only relevant information in the visualizations.

> I think since it's possible to go to particular directories and system parts, I don't think it [scalability] is a big problem, at least in our code base. ... So I think it's possible to navigate.
>
> - Participant 3

> I think it would would be scalable. I mean, you can search for your team, so as long as everyone has unique teams ... So, I think it would scale.
>
> - Participant 8

Preference for Simplicity When asked to evaluate the visual elements, developers frequently preferred simplistic visualizations over more detailed but complex ones. The solution conveys a lot of information in a hierarchical way: directories have files with surviving mutants. Most developers preferred a separate visualization for each level in the hierarchy over one that supports hierarchical data and can show information related to multiple levels, such as the treemap.

> The gauge is very intuitive, I guess it changes color to yellow and red the further down you come.
>
> - Participant 4

> I mean, this one is a bit hard to read for me, the Treemap. So maybe one can do something else there.
>
> - Participant 7

> I guess it [treemap] was a bit confusing then.
>
> - Participant 10

Evolution and Trends Developers frequently praised visualizations that supported showing information over time, as they considered them to enable them to see the evolution of MT results and contextualize the results more. They also valued visualizations that enabled them to see the impact of their changes.

> I mean, this one is quite good [lineplot], the surviving mutants per directory over time. You're seeing how the directory is getting affected.
>
> - Participant 1

> I mean, I like having it over time like this [lineplot].
>
> - Participant 2

> Let me think, but I think this graph [lineplot] is absolutely important to see which way your trend goes.
>
> - Participant 6

5.4.5 User Experience

In the following section, we highlight the factors developers mentioned when reflecting on the interactability of the developed solution.

Filtering by Data A feature considered important was the possibility to filter by data, something they utilized in the dashboard but lacked in the report. An example of data a user could filter was guardianship, a directory, or a specific file. This allowed developers to find the specific information they were looking for more easily and argued that it would be more overwhelming without filtering data.

> I think since we have quite a huge code base, I don't know how useful these are without filtering.
>
> - Participant 3

> However, in the report, I cannot filter by team, right? I can only go by directory.
>
> - Participant 9

> Like for all teams, yeah, it's a bit overwhelming. But I think when you got into your specific team or [team at Zenseact]. I think it was a good amount [of information].
>
> - Participant 10

Navigability While using the developed solution, participants frequently switched between the dashboard and the report. A common point brought up was that they lacked a way to switch between the two conveniently: they would find a point of interest using the dashboard and manually navigate the report to see it in more detail. They considered that a button that would have taken them from the dashboard to the same point in the report would have made the experience less cumbersome. Some also argued in favor of combining the solutions somehow into a single component.

> If I could press on the link and get it [the report] open to see it directly. So that's some kind of link from this page to this board or something. But yes, that would be nice just to click here ...
>
> - Participant 2

> But if you could create a dashboard where you include this [the report] as a part of the dashboard? ... I Navigated there [to the report], which wasn't a big issue, but it would be nice if they could combine.
>
> - Participant 8

> I mean, since they're not connected right now, I wouldn't maybe use them in combination.
>
> - Participant 10

Learning Curve Seven participants reflected that even with the initial presentation about MT at the start, it was still hard for them to use it. They attributed this both to the developed solution and MT itself. Regarding MT, they did not find the mutation score intuitive to interpret, unlike a CC score. Many mentioned wanting to see the results of more teams to get a notion of what a "good" mutation score could be. Regarding the solution, they attributed most of their issues to interactive elements not being straightforward and requiring more usage to know what is possible and what isn't.

> There's definitely a learning curve. That's with everything that we do, you need to work with it to know the flaws and so on, I think.
>
> - Participant 2

> It's hard to assess the quality without comparing it to others.
>
> - Participant 4

> I mean, it was quite clear. I mean, once you get that you will add filters by clicking a lot of things, it makes more sense. Because I don't really know in the beginning what will happen if I click this ... But as soon as you understand that you will work with filters in this dashboard, I think it's clear.
>
> - Participant 7

> But I feel like it is a bit hard to know what is a good value, or percentage. I guess there would be guidelines for the repo, like code coverage.
>
> - Participant 10

Usability Many participants reflected positively on the developed solution after using it, explaining that it was easy to use after some guidance and becoming more familiarized.

> It's quite easy to track it down, I mean, the navigation to get to this point was quite easy.
>
> - Participant 1

> It was actually great getting some hands-on experience and it was easier than I expected to use it. ... Like when you first hear about it seems like a complex concept, but it's really like quite simple actually, so it's easy to talk about. ... with the help of the tools, I mean. Otherwise, if you just get some kind of console output right, then that would be much harder. But it was a treat to use it.
>
> - Participant 6

> But I think after familiarizing myself with the dashboard for just a couple of minutes, it became really, really clear to me.
>
> - Participant 9

Cognitive Workload The participants did not consider the amount of information displayed overwhelming. They also considered that the ability to filter and drill down[2] the data aided this.

> I think this is fine. I mean, it's not too much, which it sometimes could be. I think this is a good level at least.
>
> - Participant 2

[2]The ability to filter the data to a desired level of detail/area of interest.

> I think it [the amount of information] was enough, yeah.
>
> - Participant 3

> It was not overwhelming. Coming in, I expected to be overwhelmed, but it was not overwhelming and it felt like a good amount, actually. Usually, it takes a while for you to learn how to use these efficiently and what to look for. But, it felt pretty good initially. ... I thought this was easier to grab right away, actually. ... It's not just dense information everywhere.
>
> - Participant 5

6

Discussion

In this study, we set out to integrate MT into the codebase of the partner company that works in a safety-critical domain. We documented the process using observations that were later converted into an experience report. We then used think-aloud sessions and interviews to see which information from MT results developers perceived as useful and how it should be presented. In the previous section, we focused on presenting the results from these activities; here, we interpret and discuss the results while providing recommendations to help developers integrate MT into their codebase and benefit from it.

6.1 Experience Report Conclusions (RQ1)

Several activities were carried out, and decisions were made that are worth highlighting due to their positive impact on the integration process.

The first recommendation is to make a simple proof-of-concept project to apply candidate MT tools. This previews the tool's features, workflow, and compatibility, which can be used to evaluate what tool to use. It furthermore allows developers to determine whether certain a tool's behaviour is caused by the tool itself or the codebase they want to integrate it into. As we followed this when attempting to integrate Dextool into the final codebase, we found an error that we knew was related to the tool and not the codebase, likely saving us a significant amount of time we would have spent investigating the problem.

In hindsight, we also consider that by trying out the tool early on, you can already observe workflow, nuances, and limitations of the tool. Certain issues that will be encountered when integrating into the final codebase can already be foreshadowed by experimenting with the tool on a proof of concept. However, it can be difficult to realize the issues when experimenting, especially if the developer is unfamiliar with MT. Nevertheless, initial experimentation gives the developer a feeling of how the tool behaves, allowing them to make better design decisions when integrating it into the final codebase.

RQ1

Challenge 1
Mutation testing tools can have nuanced behavior and be error-prone, which becomes problematic during integration.

Recommendation 1
Take adequate time to assess the mutation testing tool early in a simple codebase to better understand the tool and its limitations before integrating it into your codebase.

One issue we encountered was that the MT testing tool ran the same mutant multiple times since it generated mutants for source code dependencies. We disabled this, which removed redundancy and improved the MT tool's execution time. Given the contextual factors, we considered this the appropriate decision. However, completely different scenarios might be encountered in a different context, such as with a different tool. Given your codebase and MT tool, unique issues will likely be encountered that must be addressed in one way or another. To do so, knowledge of the codebase, the end goal of MT, and the tradeoffs should be considered to handle the issue appropriately.

RQ1

Challenge 2
The combination of the tool and codebase created unique situations and issues that had to be addressed.

Recommendation 2
Plan with contingency time to handle unique dilemmas given your codebase and mutation testing tool, for which you must deliberate on different solutions given your context.

An important aspect of integrating an MT tool is the required knowledge. During the integration carried out in this study, particular knowledge of what MT was and how it worked exactly was not required: knowing what a surviving or killed mutant was did not assist in integrating the actual tool. Instead, knowledge of the MT tool, build system, and CI pipeline infrastructure was more important in completing the integration. By involving developers with knowledge about said systems, they were able to offer their guidance and expertise to make the right decisions. We conclude that integrating MT likely requires involvement from people not necessarily responsible for quality assurance. These could be people from the build system, CI, or infrastructure teams.

RQ1

Challenge 3
Integrating a mutation testing tool requires knowledge other than about mutation testing.

Recommendation 3
Involve experts from various areas, such as Continuous Integration (CI), build systems, and tool maintenance teams, so they can offer guidance when integrating a mutation testing tool.

During the integration, a very incremental workstyle was adopted. We started out using only a small subset of mutation operators and testing targets, which was gradually increased. This enabled a quick feedback loop which enabled finding and addressing issues quickly as they appeared. For instance, we found hardware-specific test targets and mutation operators to cause tool malfunctions without an extensive investigation of what was occurring. We followed an agile strategy when working locally but not in CI, which proved to be a friction point. A significant amount of time had to be dedicated to solving issues in CI since it had a long feedback loop, and it was difficult to find the issue since all pipeline code was added to CI simultaneously in a single commit instead of iteratively.

RQ1

Challenge 4
Errors occurred when all code was added to CI at once, which made it hard to debug.

Recommendation 4
When integrating a mutation testing tool, work and verify incrementally and in an agile fashion, both locally and in CI.

One of the big focuses of this project was to create something that ended up providing value to the company. As such, we intended to integrate a tool that was both fast and easy to integrate, to then focus on presenting the results in a tangible and actionable way. Mull showed signs of being one of the easier tools to integrate. Yet, we had to dedicate a significant amount of time to integrate it regardless due to the various complications we had encountered. As a result, a significant amount of code dependent on the tool had to be developed to be able to use the tool as we intended.

We consider the amount of work that had to be put into integrating the tool to be unreasonably excessive: adding a new tool should, optimally, be swift and without having to adapt to the codebase. It is unrealistic to expect developers to invest significant time integrating an MT tool until they start getting value from it when they wish to try it out. Furthermore, the MT tool is another component of the codebase that evaluates the quality of the tests, which is likely not the highest priority to managers or developers in a fast-paced environment. As is, we consider the integration process of MT tools still problematic and would expect many developers

to be dissuaded from attempting it or giving up.

This result aligns with findings from Vercacmmen et al. [16], which reports 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.".

RQ1

Challenge 5
To integrate the mutation testing tool and ensure its proper function, a significant amount of code had to be developed.

Tool Recommendation 1
Focus on the flexibility of mutation testing tools so they can more seamlessly be integrated into a software stack and workflow as intended.

After the challenging integration process, a solution was made that provided value to developers at the partner company. The report and the developed dashboard were positively received by developers, who then used them to improve their test suite. If the developed MT pipeline were to break, it is more likely to be fixed if a user finds out and notifies those responsible. On the other hand, without providing value, the solution would not be used and would slowly deprecate and break, without the tool's maintainer being aware. We consider the regular use of results by developers to be an essential part of a successful and complete integration process.

RQ1

Challenge 6
An integrated tool is always at the risk of breaking, losing the benefit it was providing, and wasting the effort to integrate it in the first place.

Recommendation 5
Plan and carry out the integration process without leaving it in an intermediary state. By the end, value must be provided to your developers or the integration will likely be deprecated.

6.2 Developer Perception Conclusions

During the think-aloud sessions and interviews, developers made many statements that gave insights into how they wanted to use MT, who should be using it, which information they wanted, how the information should be presented, and how beneficial the results were. Here, we highlight interesting findings that we argue should be considered when maximizing the benefits of using MT.

6.2.1 Mutation Testing Workflow (RQ2)

When discussing the benefits of MT, the focus is often on addressing surviving mutants to improve test suites. Less focus is given to the fact that MT enables an overview of the current state of the test suite. During this study, developers also appreciated MT as a way to obtain a good overview of the test suite quality. The findings of this project highlight that there should be a bigger focus on using MT to track the technical debt related to the test suite, compared to addressing surviving mutants. This furthermore guides developers to find out where in the test suite they should focus on and improve. To enable this, solutions such as the presented dashboard can be developed. This minimizes a developer's time and effort in getting the information of interest. A lesser finding was that developers speculated that code with many mutants suggests that said code could be refactored to improve the source code quality.

RQ2

Finding 1
Providing an overview of the mutation testing results in a dashboard enabled developers to more easily find what required improvement the most.

Recommendation 6
Getting an overview of mutation testing results is also important, instead of only seeing surviving mutants in the source code.

Mutation testing is usually framed as a tool for just developers. The results suggest that MT results can be presented in a way that does not require in-depth knowledge of the codebase, such as in the developed dashboard. Consequently, the solution can be used by professionals in more managerial positions and teams to get an overview of the state of tests in the codebase and plan their next steps accordingly. On the other hand, developers still prefer to focus on more granular data, as presented in the report.

RQ2

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

Recommendation 7
Mutation testing results can also be presented in a way that does not require knowledge of the code to benefit non-technical people, using aggregated scores (such as score per team) instead of specific surviving mutants.

We evaluated presenting results from MT, which are traditionally not shared. For instance, we highlighted mutation operators and the number of surviving mutants more to see whether developers perceived them as useful. The general consensus reached was that the only information from MT that the developers perceived as

useful were the mutation score and each surviving mutant in the actual source code, which is already what is usually presented by MT tools.

RQ2

Finding 3
Developers were only interested in mutation scores and the number of surviving mutants in the source code. They were not interested in mutation operators or the number of surviving mutants.

Recommendation 8
From mutation testing tools, focus on presenting the mutation score and surviving mutants over other information the tool provides.

One of the project's more important discoveries was made when asking participants which additional information they thought would have assisted them. Consistently, participants mentioned wanting more contextual information related to the codebase, which brought us to the following observation. As is, MT is its own standalone tool in a codebase, and it is uncommon for its result to be associated and enhanced with other tools.

Using MT results and contrasting them with other information sources can significantly enhance the information MT conveys. For instance, participants requested information on code coverage, the safety-critical standard that applied to the code, the lines of code of the file, and more. Code coverage could indicate whether a test should be modified or created, depending on whether the area has any coverage or not. As-is, we consider MT to be lacking in the sense that it is a standalone tool and could instead be better used in conjunction with other tools and information.

RQ2

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

Recommendation 9
Combine mutation testing information with other data, such as line coverage, to get a more detailed view of test quality.

Another finding, which has also been encountered in other research [14, 15], is that developers want more context on the surviving mutants. Requests were made to see the severity and the effort to fix a given surviving mutant. The participants reasoned this was important to decide how to prioritize addressing specific surviving mutants. Nonetheless, applying the advice from the previous paragraph, a developer could experiment with prioritizing surviving mutants based on information sources other than MT, such as using code criticality.

RQ2

Finding 5
Developers wanted more information that helps prioritize addressing surviving mutants.

Tool Recommendation 2
Provide contextual information of surviving mutants, enabling developers to address and prioritize them.

Recommendation 10
Attempt to combine surviving mutants with other contextual factors (such as severity) deemed important to be able to address and prioritize them.

6.2.2 How to Present Mutation Testing Results (RQ3)

How results were presented was also an important aspect of MT. If the results are poorly presented and require significant effort to interpret, users are unlikely to use them regardless of how useful they could be. Developers found multiple factors important when reflecting on how information was presented in the solution. These included the scalability and simplicity of visualizations and the ability to observe evolution over time.

RQ3

Finding 6
Developers found scalability, simplicity of visualizations, and evolution over time to be important aspects of interpreting mutation testing result visualizations.

Recommendation 11
When deciding between different data visualizations to show mutation testing results, take into consideration simplicity, scalability, and the ability to show evolution over time.

The usability of the tool used to present the tools was also an important aspect for the developer's experience. In this case, many participants found interactivity to be an essential feature that they used, which can be split up into two aspects that go hand in hand. The first aspect is that of filtering: developers only wanted to see the information that was relevant to them. As such, they considered the ability to filter by guardianship a very important feature, as it allowed them to quickly see the MT results of only their team while hiding information about other teams that was not relevant to them.

RQ3

Finding 7
Mutation testing results provide much information about a codebase, but only a fraction is relevant to a developer.

Recommendation 12
Allow filtering the results to be scoped to an area of interest, such as per team, directory, or file.

The second aspect was navigation. As previously described, MT results are hierarchical: teams have directories that have files. Participants found it important to navigate into different levels of the hierarchy quickly. In the developed solution, for instance, they could navigate from a mutation score for the entire codebase to the surviving mutants of an individual file in the codebase.

RQ3

Finding 8
Developers want to navigate through the mutation testing results of the codebase, which is hierarchical.

Recommendation 13
Allow navigation that allows scoping the data to a desired level of granularity, to explore the results easily.

Finally, it should be highlighted that MT is still a novel technique with which many developers are unfamiliar. Many developers reflected that they needed to use the solution more to understand what a good mutation score could be. This hints that a user guide and company guidelines on how to work with MT would assist developers in getting over the learning curve and thus aid in adopting MT.

RQ3

Finding 9
Most developers are still unfamiliar with mutation testing and require guidance to use the provided solution successfully.

Recommendation 14
Mutation testing is not widely used, and guidance is likely needed. To aid adoption, perform a workshop or create a user guide.

6.3 Final Recommendations

Table 6.1 presents a list including all previous recommendations.

Table 6.1: Mutation testing recommendations.

ID	Recommendation
	RQ1
R1	Take adequate time to assess the mutation testing tool early in a simple codebase to better understand the tool and its limitations before integrating it into your codebase.
R2	Plan with contingency time to handle unique dilemmas given your codebase and mutation testing tool, for which you must deliberate on different solutions given your context.
R3	Involve experts from various areas, such as Continuous Integration (CI), build systems, and tool maintenance teams, so they can offer guidance when integrating a mutation testing tool.
R4	When integrating a mutation testing tool, work and verify incrementally and in an agile fashion, both locally and in CI.
R5	Plan and carry out the integration process without leaving it in an intermediary state. By the end, value must be provided to your developers or the integration will likely be deprecated.
	RQ2
R6	Getting an overview of mutation testing results is also important, instead of only seeing surviving mutants in the source code.
R7	Mutation testing results can also be presented in a way that does not require knowledge of the code to benefit non-technical people, using aggregated scores (such as score per team) instead of specific surviving mutants.
R8	From mutation testing tools, focus on presenting the mutation score and surviving mutants over other information the tool provides.
R9	Combine mutation testing information with other data, such as line coverage, to get a more detailed view of test quality.
R10	Attempt to combine surviving mutants with other contextual factors (such as severity) deemed important to be able to address and prioritize them.
	RQ3
R11	When deciding between different data visualizations to show mutation testing results, take into consideration simplicity, scalability, and the ability to show evolution over time.
R12	Allow filtering the results to be scoped to an area of interest, such as per team, directory, or file.
R13	Allow navigation that allows scoping the data to a desired level of granularity, to explore the results easily.
R14	Mutation testing is not widely used, and guidance is likely needed. To aid adoption, perform a workshop or create a user guide.
	Tool Recommendations
TR1	Focus on the flexibility of mutation testing tools so they can more seamlessly be integrated into a software stack and workflow as intended.
TR2	Provide contextual information of surviving mutants, enabling developers to address and prioritize them.

6.4 Threats to Validity

This section covers the identified validity threats categorized into construct, internal, external, and conclusion validities.

6.4.1 Conclusion Validity

Although conclusion validity is commonly associated with the correct statistical analysis of quantitative data to make valid conclusions, it is still relevant in a qualitative study.

The think-aloud and interview sessions were performed with 10 participants, which could be considered insufficient to reach a conclusion. However, during the thematic analysis, it was deemed that theoretical saturation had been reached; as with the last participants, no new and unique insights were gained. Instead, the last participants backed the claims made by previous candidates.

6.4.2 Internal Validity

With the convenience sampling strategy of only using participants from the partner company, a potential risk is having like-minded people and biased results. However, purposive sampling within the partner company was applied to mitigate this. To do so, a well-connected employee at Zenseact was consulted to suggest people with various interests and responsibilities, weakening the single group threat. The employee had been in the company since its foundation and, at the time, worked as a technical expert in safety-critical C++ code. Because of this, the employee worked closely with all C++ software development teams and had vast knowledge about suitable participants with diverse backgrounds.

RQ1 is answered by creating observations while integrating the MT tool. As we (the researchers) perform the integration and observations, this is a possible internal validity threat. The threat is due to cognitive biases and the possibility of less rigorous protocols than when studying subjects. However, we argue this was mitigated by our observation protocol, where we specified what aspects to observe. Furthermore, developers at Zenseact have also participated in taking and discussing observations, reducing the amount of bias.

There is also a risk of human-related factors affecting the results of RQ2, RQ3. For instance, participants may have altered answers during the think-aloud and interview sessions since they were being observed. Furthermore, a developer might be untruthful and overoptimistic during the interview due to social desirability bias, which causes participants not to share their actual opinions to appear more likable to the researchers. The participants were generally also inexperienced with MT and could have had different impressions from developers with more MT experience. Another potential influence is the people's resistance to change. Lastly, there might be some preconceived biases depending on the organizational culture.

Reflexive thematic analysis views the researcher's subjectivity as a resource [2] and is a crucial part of the coding process. However, this subjectivity can be seen as a bias. Nonetheless, the authors argue that it is part of the process. And, an agreement was reached between the two coders during the thematic analysis.

6.4.3 Construct Validity

Mutation testing is relatively uncommon in the industry, and a construct validity threat is the interviewee's lack of understanding of the needed concepts. The think-aloud sessions and interviews contain a quick overview of MT to ensure the interviewees understand the concepts. However, with little prior information, the interviewees may not understand all the information. During various phases of the think-aloud and interview, the participant was reminded that they could ask for clarification at any time.

Another possible construct validity threat is because of ambiguous topics such as "useful", "information", and "visualization". However, we consider this to be less problematic since, during the interviews, we asked for clarification when we considered the participant's answer ambiguous. Furthermore, we also explicitly stated the difference between information and visualizations. When participants gave responses deemed ambiguous, they were also requested to clarify their thoughts.

6.4.4 External Validity

The case study was conducted in a unique context composed of the partner company, its software stack, its codebase, the MT tool integrated, the mutation operators of the tool used, and the tools used to present the results. Organizational and domain-related factors could also have affected the results due to safety-critical code and safety standards applied to them, for instance.

This unique but realistic setting reduces the generalizability of our findings, which is inherent in a case study. Due to this, the recommendations presented in this study are likely not universally applicable. We do argue, however, that the findings of this study can still be generalized to similar contexts, such as safety-critical domains and companies similar in size to Zenseact.

An effort was made to make the study's recommendations agnostic from the context to improve generalizability. However, we recommend that you interpret each recommendation, considering the unique context in which you, as a developer, are.

7

Conclusion

This thesis consists of research to explore the challenges encountered when integrating a mutation testing tool in an industry setting and how to address them. It furthermore focuses on what information from mutation testing results developers perceive as useful and how said results should be presented.

A case study was performed at Zenseact, a company developing safety-critical C++ code in the automotive industry. A mutation testing tool was integrated into their codebase, during which systematic observations were conducted and transformed into an experience report. The experience report reflects on the challenges encountered during the integration process and how they are addressed. This process resulted in mutation testing being performed daily in a CI pipeline, the results of which could be seen using an HTML report and a developed Kibana dashboard.

The results of this study provide insights and guidance to facilitate the adoption of mutation testing in industry settings and maximize the benefits obtained. Our contributions include recommendations regarding what to consider when integrating mutation testing tools into a codebase and recommendations regarding what information should be presented to developers and how. A table including all recommendations resulting from this study is presented in Table 6.1.

Another focus of the study is on maximizing the benefits developers can obtain from mutation testing by evaluating what information from mutation testing results they perceive as useful and how it should be visualized. To do so, think-aloud sessions and semi-structured are carried out with ten developers from the partner company, focusing on what they perceive as useful from the report and dashboard. The results are then analyzed with reflexive thematic analysis.

Our findings include that integrating mutation testing tools still requires significant effort for it to function as intended in the given codebase. To combat this, we recommend integrating iteratively both locally and in CI, to address emerging challenges. Our findings also show that to integrate mutation testing tools, it is beneficial to involve experts in other areas of expertise, such as build-system engineers, tool maintainers, and CI engineers.

With regard to what information was perceived as useful, developers preferred simple

information, such as surviving mutants and the mutation score. Instead of wanting more information related to mutation testing, they instead wanted to enrich the mutation testing results by adding other contextual information from the codebase, such as line coverage, code criticality, and code complexity.

Presenting the results with interactive elements such as filtering and scoping to an area of interest, such as a team, directory, or file, enabled developers to conveniently find the information they were looking for and gave them the perception that it was scaleable. Developers also concluded that getting an overview of mutation testing results was beneficial to them, in addition to a more detailed view, which they preferred to use to address issues in the testing suite. Additionally, we observe that mutation testing has a learning curve and requires guidelines when used.

The results generally indicate that mutation testing is a standalone technique that can be used in a codebase ecosystem. To make mutation testing more viable, mutation testing tool developers should focus on making their tools easier to integrate and easier to make them behave in the way intended by the developer. Furthermore, mutation testing tools should integrate more with other aspects of the codebase to give more holistic results by including information such as line coverage.

7.1 Future Research Directions

In this study, we evaluated both information and visualizations related to MT using qualitative analysis, focusing on what the developers perceived as useful. However, there may be a disconnect between what a developer perceives as useful and what actually is useful. The recommendations presented in this study could be re-evaluated using a more quantitative and longitudinal approach, measuring improvements in the test suite and, thus, the effectiveness of the different recommendations.

It is also important to note that developers gave many suggestions for improving their developer experience that had potential. Implementing their recommendations and repeating the think-aloud sessions and interviews would be worthwhile to strengthen the findings and possibly reveal more.

The study also focused on how developers perceived the MT results, but they, in turn, mentioned that professionals in more managerial positions could benefit from MT results, such as product owners and team managers. As such, repeating the think-aloud sessions and interviews in said roles could give insights into whether they could take advantage of MT results.

Finally, it would be worthwhile to validate these findings in other contexts. For RQ1, a specific tool was integrated into a specific software stack (programming language, CI server, build tool, visualization tool...), leading to fairly specific situations. By varying the technologies used, results could be contrasted. For RQ2, RQ3, the population used was that of developers working in the safety-critical automotive industry, so it would also be compelling to repeat this project in a different setting. This, in turn, could validate our findings in different contexts.

Bibliography

[1] Elastic, "Overview Dashboard - Elastic Security Solution." `https://www.elastic.co/guide/en/kibana/7.17/create-a-dashboard-of-panels-with-web-server-data.html`, 2023. [Online; accessed 05-May-2024].

[2] V. C. Virginia Braun, *Thematic Analysis - A practical guide*. SAGE Publications Ltd., 1st ed., 2021.

[3] H. Zhu, "A formal analysis of the subsume relation between software test adequacy criteria," *IEEE Transactions on Software Engineering*, vol. 22, no. 4, pp. 248–255, 1996.

[4] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*, vol. 112, pp. 275–378, Elsevier, 2019.

[5] A. Parsai and S. Demeyer, "Comparing mutation coverage against branch coverage in an industrial setting," *International Journal on Software Tools for Technology Transfer*, vol. 22, no. 4, pp. 365–388, 2020.

[6] N. Li, M. West, A. Escalona, and V. H. Durelli, "Mutation testing in practice using ruby," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 1–6, IEEE, 2015.

[7] A. Garg, M. Ojdanic, R. Degiovanni, T. T. Chekam, M. Papadakis, and Y. Le Traon, "Cerebro: Static subsuming mutant selection," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 24–43, 2022.

[8] W. Ma, T. Laurent, M. Ojdanić, T. T. Chekam, A. Ventresque, and M. Papadakis, "Commit-aware mutation testing," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 394–405, IEEE, 2020.

[9] M. Ojdanic, E. Soremekun, R. Degiovanni, M. Papadakis, and Y. Le Traon, "Mutation testing in evolving systems: Studying the relevance of mutants to code evolution," *ACM Transactions on Software Engineering and Methodology*,

vol. 32, no. 1, pp. 1–39, 2023.

[10] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *IEEE Transactions on Software Engineering*, vol. 45, no. 9, pp. 898–918, 2019.

[11] R. Degiovanni and M. Papadakis, "μbert: Mutation testing using pre-trained language models," 2022.

[12] G. Gay, M. Staats, M. Whalen, and M. P. E. Heimdahl, "The risks of coverage-directed test case generation," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 803–819, 2015.

[13] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[14] G. Petrovic, M. Ivankovic, B. Kurtz, P. Ammann, and R. Just, "An industrial application of mutation testing: Lessons, challenges, and research directions," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 47–53, 2018.

[15] M. Beller, C.-P. Wong, J. Bader, A. Scott, M. Machalica, S. Chandra, and E. Meijer, "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)*, pp. 268–277, IEEE, 2021.

[16] S. Vercacmmen, M. Borg, and S. Demeyer, "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)*, pp. 334–343, IEEE, 2023.

[17] M. Fowler and M. Foemmel, "Continuous integration," *martinfowler.com*, 2006. Accessed: May 21, 2020. [Online]. Available: https://tinyurl.com/ycbl2uhj.

[18] cppreference.com, "Undefined behavior," 2024. [Online; accessed 23-February-2024].

[19] J. Örgård, G. Gay, F. G. de Oliveira Neto, and K. Viggedal, "Mutation testing in continuous integration: An exploratory industrial case study," in *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 324–333, IEEE, 2023.

[20] E. R. Tufte, *The visual display of quantitative information*, vol. 2. Graphics press Cheshire, CT, 2001.

[21] Y. Adler, N. Behar, O. Raz, O. Shehory, N. Steindler, S. Ur, and A. Zlotnick,

"Code coverage analysis in practice for large systems," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 736–745, 2011.

[22] M. Ivanković, G. Petrović, R. Just, and G. Fraser, "Code coverage at google," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 955–963, 2019.

[23] O. Elazhary, C. Werner, Z. S. Li, D. Lowlind, N. A. Ernst, and M.-A. Storey, "Uncovering the benefits and challenges of continuous integration practices," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2570–2583, 2022.

[24] S. Berner, R. Weber, and R. K. Keller, "Enhancing software testing by judicious use of code coverage information," in *29th International Conference on Software Engineering (ICSE'07)*, pp. 612–620, IEEE, 2007.

[25] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, pp. 131–164, 2009.

[26] S. Baltes and P. Ralph, "Sampling in software engineering research: A critical review and guidelines," *Empirical Software Engineering*, vol. 27, no. 4, p. 94, 2022.

[27] Zenseact, "Zenseact linkedin post." `https://www.linkedin.com/posts/zenseact_today-eid-al-fitr-marks-the-end-of-ramadan-activity-7183796792193413120utm_source=share&utm_medium=member_desktop`, 2024. Accessed: 2024-04-16.

[28] I. S. . Committee, "Road vehicles — Functional safety," standard, International Organization for Standardization, Geneva, CH, December 2018.

[29] M. Fowler, "Code ownership." `https://martinfowler.com/bliki/CodeOwnership.html`, May 2006. Accessed: 2024-03-21.

[30] E. NV, "Kibana." `https://www.elastic.co/kibana`, 2024. Accessed: 2024-03-25.

[31] E. NV, "Kibana vega." `https://www.elastic.co/guide/en/kibana/current/vega.html#vega`, 2024. Accessed: 2024-03-28.

[32] R. Rampin and V. Rampin, "Taguette." `https://www.taguette.org/`. Accessed: May 7, 2024.

[33] K. Krippendorff, *Content analysis: An introduction to its methodology*. Sage publications, 2018.

[34] D. R. Hancock, B. Algozzine, and J. H. Lim, *Doing case study research: A*

practical guide for beginning researchers. Teachers College Press, 2021.

A

Mutation Operators

To answer RQ2, RQ3, the mutation operators in Table A.1 were applied to Zenseact's codebased to generate the results used during the think-aloud and interview sessions.

Table A.1: Mull mutation operators used.

Operator Name	**Description**
`cxx_pre_inc_to_pre_dec`	Replaces ++x with –x
`cxx_post_inc_to_post_dec`	Replaces x++ with x–
`cxx_minus_to_noop`	Replaces -x with x
`cxx_add_to_sub`	Replaces + with -
`cxx_sub_to_add`	Replaces - with +
`cxx_mul_to_div`	Replaces * with /
`cxx_div_to_mul`	Replaces / with *
`cxx_rem_to_div`	Replaces % with /
`cxx_eq_to_ne`	Replaces == with !=
`cxx_ne_to_eq`	Replaces != with ==
`cxx_le_to_gt`	Replaces <= with >
`cxx_lt_to_ge`	Replaces <with >=
`cxx_ge_to_lt`	Replaces >= with <
`cxx_gt_to_le`	Replaces >with <=
`cxx_le_to_lt`	Replaces <= with <
`cxx_lt_to_le`	Replaces <with <=
`cxx_ge_to_gt`	Replaces >= with >
`cxx_gt_to_ge`	Replaces >with >=

B

Observation Protocol

The observation protocol was created based on guidelines by Hancock et al. [34].

The information included in an observation is as follows:

- Date and time.
- Location.
- Name of person conducting the observation.
- Names of people participating in the observation.
- Event: Condition that made us start an observation, which could be one of the following:
  - Decision: An important implementation decision is taken.
  - Issue: An unexpected issue is encountered that has to be addressed.
  - Solve_issue: We address an issue previously encountered.
  - Doubt: More information is required to make an informed decision or to progress.
  - Progress: One of the work packages we defined to integrate MT is completed.
  - Sentiment: We have an opinion about some aspect related to the MT integration.
  - Hindsight: We believe we could have made a better decision in the past.
  - Discovery: We acquire new information that we can take to our advantage.
  - Meeting: We reflect on the information exchanged during a meeting with involved developers, academic supervisors, or industry supervisors.

  - End_of_week: We reflect on the events of the current week. This event was always carried out on Friday.
  - Finish_integration The reflections that were made after the integration was deemed finished.

- Observation: Here, we address the event that took place in detail.
- Next steps: An optional field where we added any activities we should carry out, given the event.

The following guidelines were also followed:

- When any of the events occur, they should be written down as soon as possible to prevent forgetting any important details.
- If unsure whether an event that took place is relevant, it should still be written down.
- Observations that are later deemed not relevant can be removed. Observations can also be modified if necessary to clarify or add information.
- The observations must be saved in sequential order.
- Each observation should start with an outline of the event. Afterwards, more detail should be added.
- New event types can be added if deemed necessary.

C

Think-aloud Session Protocol

C.1 Introduction

The participant is first informed that they will participate in a Master's Thesis project related to testing sufficiency. They were informed that their contribution could assist developers at Zenseact in improving their tests. They are then asked to give verbal consent to us to record their voice and screen. Furthermore, they are informed that any data published is anonymized and that they can retract their consent at any time by contacting the researchers.

C.2 Presentation

Afterwards, the participant is given a presentation with supporting slides about MT. This ensures they have the required knowledge to carry out the think-aloud session. During multiple points in the presentation, the participant is asked whether they wanted to ask any questions.

The presentation starts out by giving a motivating example where a test suite has a high percentage of line coverage but misses edge case values that should be tested. Then, a bug is introduced to demonstrate how the test suite would not detect the bug, proving that the test suite is indeed insufficient.

Introducing a bug and checking whether the test suite can detect it is used to explain mutation testing. Specifically, the concepts of mutants, killed mutants, surviving mutants, mutation operators, mutation score, and mutation testing tools are explained. Other concepts, such as equivalent mutants, were not explained since they were not relevant to the project's focus.

C.3 Initial questions

After the initial presentation, a set of questions were asked about the background of the developer:

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?

C.4 Think-aloud steps

They are informed that MT was applied to Zenseact's codebase and asked to carry out a set of tasks and interact with the results. They are asked to share their thoughts at all times while carrying out their tasks. They were not informed on how to carry out any given tasks, although they could ask for clarification and guidance during the think-aloud. While carrying out the tasks, one of the researchers focused on keeping them talking if necessary, whilst the other focused on resolving doubts the participant had and solving any unexpected issues that emerged.

To interact with the results, the participant is given a computer that shows both the HTML report and Kibana dashboard on the screen simultaneously. They are first given a short demonstration on how to use the essential features of the report and dashboard.

They were asked to carry out the following tasks, advancing to the next step once they felt they sufficiently addressed the current one:

1. Familiarize yourself with the dashboard and report.
2. Assess the quality of tests for team *Alpha*[1].
   (a) In which direction is the quality of the tests of the team evolving?
   (b) Is the quality of the tests for the different directories the team maintains around the same? Are there any directories which require more extensive testing?
3. Assess the quality of directory *Beta*.
   (a) In which direction is the quality of the tests of the directory evolving?
   (b) Is the quality of the tests for the different files the directory maintains around the same? Are there any files which require more extensive testing?

[1] *Alpha*, *Beta*, and *Gamma* represent a team, directory, and file that existed within Zenseact but have all been renamed here for confidentiality. The same team, directory, and file were used for all participants.

4. Assess the quality of file *Gamma*.

   (a) Are the surviving mutants concentrated around the same region of the file, or are they spread out?

   (b) If you could make or modify a test for this file, what would you address?

   (a) In which direction is the quality of the tests of the file evolving?

When they finish all tasks, they can explore MT results for their own team or teams they are interested in. Afterwards, the think-aloud activity is considered finished.

D

Interview Instrument

D.1 Introduction

The interview is performed with each participant of the think-aloud session, after a short break. Before starting, the participant is reminded that the consent for recording they gave during the think-aloud session still applies during the interview.

Afterwards, the interviewers distinguish the concepts of information (RQ2) and visualizations (RQ3). The participant is also informed that they can ask for clarification about a question at any time, and that all questions apply to both the HTML report and the Kibana dashboard. The interviewees also ask follow-up questions for clarification (e.g., "Why?").

D.2 Questions

D.2.1 Questions relating to information (RQ2)

1. Did you find any information particularly useful?
2. Did you find any information particularly not useful?
3. Did you find any useful information missing?

D.2.2 Questions relating 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 useful information missing?
4. What was your experience with the amount of information provided in both the report and dashboard?
5. What was your experience looking for the information you needed to look at,

given your tasks?

6. Do you think the experience of using the report & 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?

D.2.3 Concluding questions (RQ2 and RQ3)

1. Do you believe the potential benefits of mutation testing are worth the time and effort required to learn how 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 we gave 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?

D.2.4 Wrap-up

The participant is first informed of the study's exact goal and how CI was applied. They are then thanked for their time and asked not to share their impressions with any future participants, in order to not bias them.