



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

---

# **Recommendations for Mutation Testing as Part of a Continuous Integration Pipeline**

With a focus on C++

Master's thesis in Computer Science and Engineering

**JONATHAN ÖRGÅRD**



MASTER'S THESIS 2022

# Recommendations for Mutation Testing as Part of a Continuous Integration Pipeline

With a focus on C++

JONATHAN ÖRGÅRD



UNIVERSITY OF  
GOTHENBURG

---



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering  
CHALMERS UNIVERSITY OF TECHNOLOGY  
UNIVERSITY OF GOTHENBURG  
Gothenburg, Sweden 2022

Recommendations for Mutation Testing as Part of a Continuous Integration Pipeline  
With a focus on C++  
JONATHAN ÖRGÅRD

© JONATHAN ÖRGÅRD, 2022.

Supervisor 1: Gregory Gay, Department of Computer Science and Engineering  
Supervisor 2: Francisco Gomes de Oliveira Neto, Department of Computer Science  
and Engineering  
Advisor: Kim Viggedal, Zenseact  
Examiner: Daniel Strüber, Department of Computer Science and Engineering

Master's Thesis 2022  
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 L<sup>A</sup>T<sub>E</sub>X  
Gothenburg, Sweden 2022

# Recommendations for Mutation Testing as Part of a Continuous Integration Pipeline With a focus on C++

JONATHAN ÖRGÅRD

Department of Computer Science and Engineering

Chalmers University of Technology and University of Gothenburg

## Abstract

Mutation testing aims to judge the quality of a test suite by evaluating the ability of the tests to detect subtle changes in the code. Altered versions of the code called mutants are generated by a mutation tool, often a subtle change based on common syntactic mistakes. Each mutant is then run against the original test suite to see if the test results change.

This thesis aims to identify recommendations to mitigate the fact that mutation testing can be notoriously expensive, making it difficult to integrate as part of a continuous integration (CI) pipeline.

An evaluation of existing mutation testing tools for C++ was done to better understand the functionality and limitations of the available tools. Simultaneously, a case study was conducted at Zenseact to identify suggestions on how mutation testing can best be used within CI. First, a literature review was conducted to examine observations made by other studies on the application of mutation testing in practice, and to identify techniques and practices to perform mutation testing effectively. Next, stakeholder interviews were conducted to assess the developer's view of effective mutation testing. The result of the tool evaluation showed that existing mutation tools for C++ differ in their capabilities, and the case study result showed that there exist plenty of techniques to effectively integrate mutation testing into CI from the developers' perspective.

Based on the result, only the mutation tools Dextool and Mull were recommended for CI integration. Furthermore, guidelines were developed such as prioritizing mutation testing on essential areas of the code, performing mutation testing on old code only when there is free machine time, and that inexperienced developers should be trained to understand the mutation result better.

Keywords: computer science, engineering, project, thesis, mutation testing, continuous integration, C++, case study, dextool, mull



# Acknowledgements

I want to express my gratitude to my two supervisors, Gregory Gay and Francisco Gomes, for providing guidance and feedback throughout this thesis. I would also like to thank the partner company for this thesis, Zenseact. A special thanks to Kim Viggedal, my advisor from Zenseact, who was always ready to offer support or send me in the direction of someone who could. And a big thanks to everyone from Zenseact who agreed to participate in interviews or help out in any other way.

Jonathan Örgård, Gothenburg, June 2022





# Contents

<b>List of Figures</b>	<b>xiii</b>
<b>List of Tables</b>	<b>xv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem Description . . . . .	2
1.2 Purpose of the Study . . . . .	3
1.3 Significance of the Study . . . . .	3
1.4 Thesis Outline . . . . .	4
<b>2 Background</b>	<b>5</b>
2.1 Mutation testing . . . . .	5
2.1.1 Unproductive mutants . . . . .	6
2.1.2 Mutant schema . . . . .	6
2.2 Continuous Integration . . . . .	7
2.3 Containers . . . . .	7
2.4 LLVM and AST . . . . .	7
2.5 Existing C++ Mutation Tools . . . . .	8
2.5.1 MuCPP . . . . .	8
2.5.2 Mull . . . . .	9
2.5.3 Dextool . . . . .	9
2.5.4 CCmutator . . . . .	9
2.5.5 Mutate++ . . . . .	10
<b>3 Related Work</b>	<b>11</b>
3.1 Reducing the Cost of Mutation Testing . . . . .	11
3.2 Applying Mutation Testing in Practice . . . . .	11
3.3 Mutation Testing Tools . . . . .	13
<b>4 Methods</b>	<b>15</b>
4.1 Research Design . . . . .	16
4.2 Context from the Partner Company . . . . .	17
4.3 Evaluation of C++ Mutation Tools (RQ1) . . . . .	18
4.4 Usage of Tools in CI (RQ2) . . . . .	20
4.4.1 Developers' views of effective mutation testing (RQ2.1) . . . . .	20
4.4.2 Techniques to effectively meet the goals of the developers (RQ2.2) . . . . .	22

4.4.3	Guidelines for Mutation Testing Within CI (RQ2.3)	22
<b>5</b>	<b>Results</b>	<b>25</b>
5.1	Evaluation of C++ Mutation Tools (RQ1)	25
5.1.1	Installing and using the tools	25
5.1.2	Comparing the feature set of the tools	26
5.1.3	Comparing the mutation operators of the tools	27
5.1.4	Mutation generation results	28
5.1.5	Mutation execution results	29
5.1.6	Clang dependency	30
5.1.7	CI workflow implementation	30
5.2	Mutation Testing Within CI (RQ2)	31
5.2.1	Developers' Views of Effective Mutation Testing (RQ2.1)	31
5.2.1.1	Test Quality	32
5.2.1.2	Developer Impact	35
5.2.1.3	Implementation	37
5.2.1.4	When to use	39
5.2.1.5	Summary	40
5.2.2	Techniques for Effective Mutation Testing (RQ2.2)	40
5.2.2.1	Techniques for Effective Mutation from the Literature Review	41
5.2.3	Guidelines for Mutation Testing in CI (RQ2.3)	45
<b>6</b>	<b>Discussion</b>	<b>49</b>
6.1	Mutation tool summary	49
6.1.1	Dextool	49
6.1.2	Mull	49
6.1.3	MuCPP	50
6.1.4	Mutate++	50
6.1.5	CCmutator	50
6.2	Identifying tools suitable for CI integration	50
6.3	Possible tool improvements	51
6.4	Compiler Compatibility	51
6.5	Threats to Validity	52
6.5.1	Construct Validity	52
6.5.2	Internal Validity	52
6.5.3	External Validity	53
6.5.4	Reliability	53
6.5.5	Conclusion Validity	53
6.6	Future Work	53
<b>7</b>	<b>Conclusion</b>	<b>55</b>
	<b>Bibliography</b>	<b>57</b>
<b>A</b>	<b>Mutation Operators</b>	<b>I</b>
A.1	MuCPP Class-level Mutation Operators	I

A.2	CCmutator Multi-threaded Mutation Operators . . . . .	II
A.3	Mutation Operator Examples . . . . .	III
<b>B</b>	<b>Mull Mutation Operator Mapping</b>	<b>XI</b>
<b>C</b>	<b>Interview Guide</b>	<b>XIII</b>
<b>D</b>	<b>Interview Data</b>	<b>XVII</b>
D.1	Sub-theme overview . . . . .	XVII



# List of Figures

4.1	Overview of the case study process. The different shades represent different parts of our case study, such as activities (light), deliverables (darker), and answers to our RQ (medium). . . . .	16
4.2	Overview of the mutation testing CI pipeline using GitHub actions. Rectangles represent actions performed by the GitHub back-end app, and rounded rectangles represent actions performed by the worker. . .	20
5.1	Overview of the steps for the Dextool CI job. . . . .	31
5.2	Dextool mutation report example. . . . .	32
5.3	Overview of generated themes from the interviews. A rectangle represents the topic, a rhombus represents themes, and an oval represents sub-themes. . . . .	33
5.4	Technique/practice and sub-theme match map. Rectangles are sub-themes, and rectangles with rounded edges are techniques/practices from the literature or tools. Themes are color-coded; purple = quality, blue = implementation, pink = when to use, and yellow = prioritization. . . . .	45
5.5	Sub-theme relational map, this map is separate from the theme overview to reduce clutter. Themes are color-coded; purple = quality, blue = implementation, pink = when to use, yellow = prioritization, and orange = risks. . . . .	46



# List of Tables

2.1	Examples of mutation operators. . . . .	6
4.1	Case study plan according to Robson [41] as contained in [42]. . . . .	17
4.2	C++ mutation tools with versions used for the thesis. MuCPP does not have a listed version on its website. It does, however, have a “last updated” post, but it is unclear if that is when the website was updated or the tool. . . . .	18
4.3	C++ projects with size, GitHub stars, and commit used for the thesis, only C and C++ files counted. . . . .	19
4.4	Interview study participants. . . . .	21
4.5	Keywords used for literature review database searches. . . . .	23
5.1	Summary of the activities and the corresponding tools in which these activities were successful. . . . .	26
5.2	Features relevant to CI integration by the different tools. . . . .	27
5.3	Traditional mutation operators that are supported by the mutation tools. . . . .	28
5.4	The number of mutants generated per project and per tool. Mull might not report the correct total mutation amount if template functions are present. Projects with templates that might affect the mutation amount are marked with a *. Projects marked with — for MuCPP did not have the prerequisite Makefile. . . . .	29
5.5	Time (in seconds) spent generating mutants for the projects using the mutation testing tools. Projects marked with — for MuCPP did not have the prerequisite Makefile. . . . .	29
5.6	Mutation execution benchmark results for each project using Dextool and Mull. Only one test binary could be executed at a time due to limitations in Mull. The baseline column represents the time it took the test suite to complete on its own, without any tool involved. . . . .	30
5.7	Versions of Clang supported by each dependent mutation tool. . . . .	31
5.8	Description of generated themes from the interviews. . . . .	32
5.9	Mutation techniques and practices from the literature and mutation tools. . . . .	42
A.1	Class-level mutation operators supported by MuCPP. . . . .	I
A.2	Multi-threaded mutation operators supported by CCmutator . . . . .	II
A.3	Mutation operators with examples . . . . .	III

B.1 Mull operator mapping . . . . . XI

D.1 Overview of generated sub-themes from the interviews. . . . . XVIII



# 1

## Introduction

Historically, it has been common to use different structural coverage metrics to estimate how well a given set of tests covers the tested code. Line coverage has been used to measure which lines of code are executed by the tests, while branch coverage has been used to take this one step further and measure which branches in the code have been executed by tests. A branch is a point in the code where control can diverge based on a conditional expression, such as an if-statement, switch statement, or loop condition. For example, code coverage has been an important source of documentation that production code is thoroughly tested in the automotive industry. A challenge with traditional structural coverage metrics is that the simple execution of code elements may not result in thorough testing, as the conditions required to meet such metrics can often be met by many different test cases [14]. These criteria mandate that code be executed, but only lightly constrain **how** that code is executed. The resulting tests may be insensitive to subtle problems in the code, only demonstrated by a small number of inputs.

Another approach that has gained some interest is mutation testing. In mutation testing, a tool generates “mutants” of the code. A mutant is an altered version of the production code. For each mutant, the original tests for the production code are executed on the mutated version of the code. If all tests pass, the mutant survives, while if one of the tests fails, the mutant is killed. The goal is to kill the mutants. If the test suite can kill a large portion of the mutants, then it is likely that the tests are **sensitive** to small changes to the production code. This is good since that indicates that tests are good at catching incorrect behavior in the code. If a large percentage of the mutants survive, then the tests are likely to be **insensitive** to these changes. This indicates that the current tests are insufficient, that additional tests need to be created, or that the existing tests need improvement. The code alterations made when generating mutants are based on **mutation operators**—modifications based on common code structures that can be made automatically by a tool. These operators are based on common syntactic mistakes made by programmers, such as inserting  $<$  instead of  $\leq$ .

The mutation score calculates the percentage of mutants killed, the number of mutants killed divided by the total number of mutants. Some practitioners argue that using this score as an indication of the adequacy of the test is superior to using test coverage metrics in specific contexts. Inozemtseva et al. [17] say that their results

suggest that test coverage metrics are a poor indicator of the effectiveness of the test suite and that the mutation score may be a good substitute. Mutation testing has the benefit of actually evaluating the ability of the tests to detect subtle changes.

An open question is how best to integrate mutation testing into the development and testing workflow. For example, developers may work with various programming languages and often build and test code as part of a Continuous Integration (CI) process. CI is a practice developers often use to automate tasks such as building and testing code when new changes are pushed to a repository. The developer then receives feedback from automated tasks, such as test results or code coverage measurements. Mutation testing could be applied during the test execution stage of CI to assess the adequacy of the tests executed on the code, for example, being applied selectively to the code elements that have been changed in the latest update.

### 1.1 Problem Description

Mutation testing is a potentially effective way to judge the quality of test cases [17]. However, two main issues hinder the widespread adaptation of mutation testing. The fact that mutation testing can be notoriously expensive and has received less attention in certain programming languages makes it difficult to integrate existing mutation testing tools with CI.

**Notoriously expensive:** Within CI, code is expected to be built, tested, and packaged within reasonable time limits so that the developer can get rapid feedback. Mutation testing is notoriously expensive because tests have to be run for each mutant separately. Therefore, instead of running the tests once, they may run many times. Another layer of complexity caused by mutation testing is that if a mutant survives, a human often has to evaluate whether it is worth taking action on the mutant or not. If no action is worth taking, developer and computational time have been wasted. The potentially high cost of mutation testing makes the use of mutations within CI a complicated task. Some work has been done with a focus on the issue of mutation in an industrial CI setting [3, 37, 38], and they have some recommendations on how mutation testing can be used. However, they use proprietary software from their own companies, making it challenging to adopt using existing open-source tools.

**Limited attention for certain languages:** In an industrial setting, developers need practical tools to perform mutation testing in a variety of languages that are used within specific domains, including C++, which has received limited attention in current research<sup>1</sup>. Although there exist tools to perform mutation testing on, for example, C++ code, it is an open question whether the tools proposed in previous research can fit into the development workflow. The tools may not be available in the desired languages and may not be usable in a flexible manner. For example, a tool may not have a command-line interface or anything else that allows it to be run

---

<sup>1</sup>We mention C++ specifically because it is used at our partner company Zenseact. However, the same point could be made about other programming languages.

programmatically, which is required to automate the tool’s usage in a CI context. Several examinations of mutation testing have been conducted from an industrial or developer perspective; see, for example, Petrovic et al. at Google [37, 38] and Beller et al. at Facebook [3]. However, these studies use proprietary tools only for internal use, making them difficult to replicate. Nevertheless, these papers share the logic behind the tool and how they apply it to their codebases.

## 1.2 Purpose of the Study

This study aimed to contribute to the knowledge of researchers and practitioners of the applicability of mutation testing in industrial software development when used as a part of CI, with a focus on tools that support mutation of C++ code. We conducted this research in cooperation with a partner company, Zenseact, which uses C++ as its primary development language in some components.

We first evaluated the capabilities of existing C++ mutation tools, including whether or not the tools can be used in CI. In addition to this, we did a case study at Zenseact to identify suggestions on how mutation testing can best be used within CI. It is an open question as to how developers could benefit the most from mutation testing as part of their testing process. Therefore, we discussed with stakeholders at Zenseact what they see as the most effective use of mutation testing, along with a literature review, to identify practices, such as using the mutation testing result as a coverage metric. We then used the results of the interviews and the literature review to suggest practices when applying mutation testing in a CI setting.

## 1.3 Significance of the Study

The scientific contribution of this thesis is the tool comparison for mutation testing in CI. The comparison of the tools and the identified techniques from the literature review can be used to identify areas to improve existing mutation tools or inspire the development of new tools. Similarly, techniques represented in the tools but not the literature could inspire future research. According to Papadakis et al. [36], a future problem to solve is “how should we integrate mutation testing into our development process”, which we contributed to in this case study.

The practical contributions of this thesis are the comparison of existing C++ mutation tools, identified guidelines for mutation testing in CI, and proof-of-concept GitHub workflows for mutation testing in CI. We compared the feature set of existing C++ mutation tools that practitioners could use along with the guidelines to see what, if any, tools can be flexibly integrated into CI and other developer workflows. The practical challenges and solutions reported here also help practitioners interested in adding mutation testing to their CI processes. Furthermore, the guidelines could help practitioners determine how mutation testing can best be used within their CI processes.

## 1.4 Thesis Outline

- Chapter 2 presents the relevant terminology and concepts for this study.
- Chapter 3 presents findings from other studies relevant to this study.
- Chapter 4 describes the methodology used in the study.
- Chapter 5 presents the results of the study.
- Chapter 6 discusses the results of the study.
- Chapter 7 presents the conclusions of the study.

# 2

## Background

To understand how to apply mutation testing in practice best, we first have to understand the concept of mutation testing, how it can be used as part of a workflow, and how the mutation testing tools work. In this section, we will first look at the concept of mutation testing, then look at continuous integration and containers that can be used to integrate mutation testing into the workflow. After that, we will look at LLVM and AST that can be used to identify where to mutate the code, and finally, we will look at how the mutation tools work.

### 2.1 Mutation testing

Mutation testing aims to judge the quality of a test suite by evaluating the ability of the tests to detect subtle changes in the code. Mutation testing was first proposed by Lipton [24] in 1971 and then published by Lipton et al. [9] in 1978. In mutation testing, a tool generates altered versions of the code, called mutants. A mutant is usually a subtle change in the code, often based on common syntactic mistakes made by programmers. The first mutation testing tool was developed by Budd [5] in 1980. For each generated mutant, the original test suite for the original code is executed on the mutated versions of the code. We expect a test in the test suite to fail because the code has changed. If a test fails, the mutant is considered killed because it has detected it. If no tests fail and the test suite fails to detect the mutant, the mutant is considered live. Several mutants can be combined to form high-order mutants. The mutation score is the percentage of how many mutants the test suite has killed [9]. The mutation score is calculated by dividing the number of dead mutants by the total number of mutants. The mutation score is an indication that the tests are insensitive to the changes introduced by the mutants. The mutation testing process can generally be divided into three phases; code analysis, mutant generation, and execution of mutant tests.

To define how mutants should be generated, we have mutation operators. They are grammatical rules to introduce a syntactic change to the original code and, as mentioned above, are usually based on common syntactic mistakes, such as inserting a  $<$  instead of  $\leq$ . See Listing 2.1 and Listing 2.2 for examples of these. Any code change could represent a mutation operator; see Table 2.1 for some examples of mutation operators or Appendix A.3 for further examples of these and other

## 2. Background

mutation operators.

**Listing 2.1:** Original code.

```
if(a < b) {  
    return a;  
}  
return b;
```

**Listing 2.2:** Mutated code using ROR.

```
if(a <= b) {  
    return a;  
}  
return b;
```

**Table 2.1:** Examples of mutation operators.

Operator	Name	Description	Example
ABS	Absolute Value Insertion	Inserts absolute value of a numeric value	$a \rightarrow \{abs(a)\}$
AOR	Arithmetic Operator Replacement	Replaces arithmetic operator with another	$a + b \rightarrow \{a - b\}$
LCR	Logical Connector Replacement	Replaces a logical connector with another	$a \&\& b \rightarrow \{a     b\}$
ROR	Relational Operator Replacement	Replaces a relational operator with another	$a < b \rightarrow \{a \leq b\}$
UOI	Unary Operator Insertion	Inserts an unary operator on an operand	$a \rightarrow \{a ++\}$
SBR	Statement block removal	Removes a statement block	$a = 1 \rightarrow \{\}$

### 2.1.1 Unproductive mutants

Some mutants trigger the same behavior as the original code; these mutants are called **equivalent mutants** and are unproductive mutants. They might never get killed by the test suite, or it might be pointless to kill them since they cause the intended behavior. Another type of unproductive mutants is those that would not improve the test suite [39]. For example, a mutant that modifies a debugging line that prints a debug text without affecting the code’s functionality. A test could be created to catch this mutant, but doing so would add no value to the test suite. Although some techniques exist to mitigate the equivalent mutant problem, the problem is theoretically undecidable [35]. Because of this, developers might have to manually check for unproductive mutants among the live mutants to flag them as irrelevant.

### 2.1.2 Mutant schema

Mutant schema is a way to speed up mutation testing by inserting multiple mutants into the code under test in an inactive state and then activating one mutant at a time to test the mutants without the need to recompile. This is done by inserting the mutants into a form of a partially interpreted program schema [45]. A partially interpreted program schema is a program that contains identifiers called abstract entities in place of parts of the code [2]. The schema can then be instantiated to form a complete program by providing code to use on the abstract entities. Mutant schema expand on the abstract entity concept by introducing meta-operator abstract entities, an abstract entity that represents the mutations generated by one mutation operator. A mutant schema replaces parts of the code where a mutant could be inserted with a meta-procedure. These functions correspond to each meta-operator abstract entity that could be applied to that part of the code. This potentially speeds up mutation execution; instead of recompiling the project between every

mutation execution, the project can instead be compiled once and be instantiated to function as any of the mutants by having the meta-procedures instantiate one mutant at a time.

A potential problem with mutant schema is that the code to compile can increase drastically since all the code variants must be inserted at once. This can potentially cause compilation problems. The mutant schema can be split into multiple sub-mutant schemas to mitigate the compilation problem. Another potential issue with mutant schema is that code execution could be slowed down compared to the original code due to the meta-procedure function call overhead.

## 2.2 Continuous Integration

Continuous Integration (CI) is a practice that developers often use to automate tasks such as building and testing code when new changes are pushed to a repository. CI could be used to automate mutation testing tasks that give feedback on the quality of new code.

GitHub Actions is a tool that can automate software development workflows, for example, creating CI workflows [15]. GitHub Actions was used in this thesis to create proof-of-concept CI workflows.

Make is a build automation tool that can, for example, be used as a part of the CI workflow to build software from source code using a Makefile. The Makefile contains instructions for the tasks to be executed.

## 2.3 Containers

Containers provide a virtual layer on top of the host operating system, allowing multiple operating systems to run alongside each other in isolation. This is unlike traditional virtual machines (VMs), which also virtualize the hardware level, causing more overhead. Docker is the leading container solution widely accepted in the industry [43]. A study by Felter et al. [12] at IBM shows that Docker containers have a negligible overhead on I/O operations and CPU performance but a noticeable impact on network performance. Containers were used in this thesis to accommodate incompatible dependencies between the different mutation testing tools that were evaluated.

## 2.4 LLVM and AST

LLVM (low-level virtual machine) is a compiler framework designed for transparent, lifelong program analysis and transformation for arbitrary programs by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs as stated in [22]. LLVM was initially implemented for C and C++ but now supports more languages. The LLVM front-end

compiles a language, for example, Clang, that compiles C++ and other C languages [25], into LLVM IR (intermediate representation). The LLVM IR is, in turn, compiled to machine code in the back-end. Having LLVM IR between the original language and the machine language allows tools to analyze and use the compiled IR code before it is compiled into the target machine language, enabling reusable tools between different target architectures.

AST (abstract syntax tree) is a syntax tree representation of the source code. Each node in the tree represents a construct, and a mutation tool can then use this tree representation to identify nodes to modify.

## 2.5 Existing C++ Mutation Tools

A few C++ mutation testing tools exist, and this thesis evaluated five of them; MuCPP, Mull, Dextool, Mutate++, and CCMutator. The documentation of the technical details for the tools varied greatly, so not all explanations in this section are of equal detail. We assessed the features relevant to a CI context, the mutation operators provided by the different tools, and benchmarked the mutation generation and test execution with the various tools.

Most mutation tools work in three phases: code analysis, mutant generation, and mutant test execution [7]. CCMutator is the only tool that does not support the mutant execution phase.

- **Analysis:** The tool traverses the AST, LLVM IR or source to analyze the code of the given files and determines where mutation operators can be applied.
- **Mutant generation:** The mutants are generated at the designated spots using mutation operators and stored for the next phase.
- **Mutant test execution:** One mutant is applied at the time and tested against the test suite. Mutants are flagged as killed or live, depending on the test suite result. A mutation score is then calculated once all mutants have been tested.

### 2.5.1 MuCPP

MuCPP is a tool developed for mutation operator research by Delgado et al. [7] and uses the Clang API to find mutation points. It requires a Makefile to function that includes rules to clean up the project folder, compile the mutation program, compile the test suite, and a rule to execute the test suite for it to function. It is also not subject to a specific testing framework; instead, it requires that the testing framework report the mutation testing result to MuCPP in a specific format. The tool uses git to store its mutants in branches, which means that the project under test cannot be in a directory containing a git repository if MuCPP is to be used since the tool creates its own repository and branches to function. Mutation operators to be used can be specified in a config file, and the tool mutates files supplied to it as



flags; if no files are provided, then the root directory is processed.

### 2.5.2 Mull

Mull is a Clang compiler plugin developed by Denisov et al. [10] that has to be enabled while compiling and works a bit differently than most mutation tools; it does the analysis and mutation generation at build time. Mull is based on LLVM and uses LLVM IR and Clang AST APIs extensively [29]. The tool works on the LLVM bitcode level and finds and creates mutations of a program in memory. This enables Mull to generate and insert mutants as the project is built; the generated mutants are injected in an inactive state into the original programs while it is being built. Each mutant is hidden under a conditional flag used to enable each mutation, much like a mutant schema. The resulting program can be compiled into a single program that can be used to test all the mutations without having to recompile. This means that the first compilation time for the project is longer, but not having to recompile potentially speeds up the mutation testing. The corresponding mutant is activated for each mutation, and the tests are then executed to see if the mutant survives.

Mull can run the mutation testing in separate sub-processes, allowing multiple mutations to be tested simultaneously without affecting each other. A report is then generated to display the result of the mutation testing. The tool is best used in conjunction with a configuration file that specifies which files to mutate and mutation operators to use. A drawback of Mull is that it requires that only one test binary be given as input, which means that all tests being used must have been compiled into a single executable. This potentially requires the project's tests to be rewritten, while other tools support multiple test binaries and can recursively find tests given a glob pattern. A glob pattern is a way to match filenames against a pattern by, for example, using wildcards.

### 2.5.3 Dextool

What makes Dextool stand out from other mutation tools is that it has a highly customizable config file and that mutation results can be reused between runs. Generated mutants and the mutation are stored in a SQL database file, which can be reused in future runs. However, if a file has changed since the last run affected mutants are rerun. The tool has partial support for mutant schema, the mutants can be combined into schemas but do not always compile [19]. If the schema compile fails, then the mutants of the schema are instead compiled and tested individually.

### 2.5.4 CCmutator

CCmutator was developed to generate partial and high-order multi-threaded mutants by Kusano et al. [21]. The tool only supports mutation analysis and mutant generation; it cannot perform the mutant testing by itself. The tool can only apply one type of mutant operator at a time on a given file and requires the user to specify which occurrences to mutate after the mutant analysis.

### 2.5.5 Mutate++

Mutate++ is a web application that runs locally. At the time of writing, the tool is in a very early stage of development, and mutants are created at a purely syntactical level, often resulting in code that fails compilation [31]. The tool is also GUI only, and files must be added manually.

# 3

## Related Work

### 3.1 Reducing the Cost of Mutation Testing

Reducing the cost of mutation testing is an important factor in integrating mutation testing with a CI pipeline, as the reduced cost can lead to faster developer feedback. There has been work done to improve the speed of mutation testing, both in computing and in developer time. There are proposed solutions like mutant schema [45] that encode multiple mutants for a program into one metaprogram. Usaola et al. [27] expand on mutant schema by proposing a way to identify mutants generated from statements to determine which test cases reach the mutated statement. This is done to only execute these test cases, reducing the number of required executions. They also propose another way to identify infinite loops at a reduced cost by implementing a loop counter to stop execution after a specific number of iterations instead of the traditional way to stop execution after a specific amount of time.

Another technique from the literature to drastically improve the speed of mutation testing is selective mutation proposed by Offut et al. [32], which reduces the cost of mutation testing by limiting the number of mutation operators and thus reducing the generated mutants, reducing the cost. They showed that selective mutation is almost as effective as non-selective mutation testing, using the five mutation operators ABS, AOR, LCR, ROR, and UOI.

Ma et al. [26] propose another approach to mutation testing, MuDelta. Their approach uses machine learning to select the mutation operator most likely to produce a relevant mutant to the commit. As the name of the approach implies, they focus on testing the new code added in the commit rather than retesting the old code.

### 3.2 Applying Mutation Testing in Practice

According to Papadakis et al. [36], there is a growing interest in mutation testing, and a future problem to solve is “how should we integrate mutation testing into our development process”. Mutation testing has already been integrated at a small number of companies. Still, we need more varied expertise in companies from different domains to get a more general idea of how to integrate mutation testing. Although we may not be able to directly adopt how other companies have applied mutation

testing in practice, we can at least learn from their applied techniques and practices. We can then use what we have learned to apply mutation testing in our context.

A common problem encountered when looking at the literature around mutation testing is the cost, both in terms of human and computational effort. Both Petrović et al. at Google [38] and Beller et al. at Facebook [3] argue that the sheer amount of mutants that can be generated can cause an insurmountable problem. Both claim that the traditional approach to mutation testing is infeasible in their respective industrial system due to their scale and size. They do not think that one should create mutants that the test suit would likely fail or give no actionable signal to developers. However, both have shown that mutation testing can be applied in a scaled production environment. Google, in particular, has used mutation testing in production with great success. Both companies use proprietary tools only for internal use. However, in these papers, they share how they developed these tools and how they can apply them to their massive code bases.

Google, for example, only applies mutants to new lines of code if it is also covered by a test and only generates one mutant per line of code. They also only present a limited number of mutations to the developers (seven times the number of total files in the changelist). Any more might cause cognitive overhead and cause the affected developer to stop using mutation testing. Only the minimal set of tests needed to run in an attempt to kill a mutant is run. They also apply rules for which lines of code to generate mutants; for example, a function name starting with the prefix `log` is filtered out. They use a limited number of mutation operators (AOR, LCR, ROR, UOI, and SBR, see Table 2.1 for a description of these operators) and select which mutation operator to use to create mutants based on historical data, on which operators usually generate interesting mutants for given code structures. This historical data is populated by mutation results from previous executions and developer feedback. Each time a developer is presented with a mutation in their code review process, they are also presented with the option to mark it as “please fix” to indicate that it is interesting or “not useful” to indicate that it is uninteresting. They have found that these mutation operators, combined with the heuristic rules, can generate mutants for 70% of high-priority bugs. Google has generated more than 1.1 million mutants using this method.

On the other hand, Facebook has opted for a less resource-intensive solution. Instead of generating many mutants and then picking the seemingly best mutants of these like Google, they have trained their tool to create mutants inspired by real-life bugs that will have a high likelihood of surviving. They achieved this by training their tool on a data set containing real code, before and after a bug fix, from the Defects4J fault data set [20] and their own codebase. Their tool compares the code before and after the fix, to learn how to reintroduce the bugs into similar code, thus creating a faulty mutant. Another study doing something similar to Facebook is Brown et al. [4], but instead of confirming that the code applied is a bug fix, they assume that any small change is a bug fix, which it may not be.

Ramler et al. [40] performed a case study in an engineering company developing

safety-critical systems. They found that mutation testing can assess the quality of a test suite that has already achieved 100% coverage branch coverage. That mutation testing provides hints about deficiencies in test cases that are difficult to discover. However, they also note that, for mutation testing to become practically useful, the scalability problems of mutation testing must be resolved. Two new faults were found in the code with an overall effort of about half a person-year. Another study by Baker et al. [1] found that mutation testing can identify shortfalls in test cases that are too obscure to be detected by manual review and that mutation testing offers a consistent measure of test quality that peer review cannot demonstrate. They argue that their study has produced evidence that existing coverage criteria are insufficient to identify test issues, potentially because engineers are too focused on satisfying coverage goals.

### 3.3 Mutation Testing Tools

As mentioned above, both Google and Facebook use proprietary tools for mutation testing, but there are also other mutation testing tools that anyone can use. This thesis looks at five of these tools available for C++ mutation testing. Some of these tools have been scientifically studied, but there is a lack of work done in the field that compares how these tools can be used in a professional development environment. To the best of our knowledge, no studies have made an empirical comparison of the capabilities and applicability of these tools. Denisov et al. [11] for example, have written a paper on the Mull implementation details, explaining how they can use direct manipulation of LLVM IR, a low-level intermediate representation of the code, to only recompile modified fragments of IR code. However, this explanation seems outdated since the paper mentions the use of LLVM JIT, which is no longer the case according to the Mull docs [29]. Nevertheless, they mention that a comparison of mutation tools is a subject for future work, which is part of the goal of this thesis.

Kusano et al. [21] wrote a paper on the multi-threaded focused CCmutator mutation testing tool. This paper is focused on presenting the C++ multi-threaded mutations created for the tool and how they overcome the challenges presented by multi-threaded mutation testing. Several papers have also used the MuCPP mutation testing tool; however, these have also not focused on the applicability of the tool. Delgado-Pérez et al. [7] developed MuCPP to evaluate class-level mutants. They showed that class mutation operators complement traditional mutation operators to create a more robust test suite. Delgado-Pérez et al. [8] used MuCPP to show that a selective mutation approach, using a subset of class mutation operators, reduces the number of mutants with a minimal loss of effectiveness.

These papers can serve to better understand the inner workings of the tools to help us better understand why the performance of the tool was what it was when compared to others in our evaluation.

### 3. Related Work

---

# 4

## Methods

This chapter will describe the methodology used in this thesis to find the answers to the following research questions:

- **RQ1:** What are the capabilities of existing C++ mutation testing tools?
- **RQ2:** How can mutation testing be best used within continuous integration?
  - **RQ2.1:** What do developers see as the most effective use of mutations in their practice?
  - **RQ2.2:** Can we identify techniques that meet the goals of developers?
  - **RQ2.3:** Can general guidelines for the use of mutation testing within CI be identified?

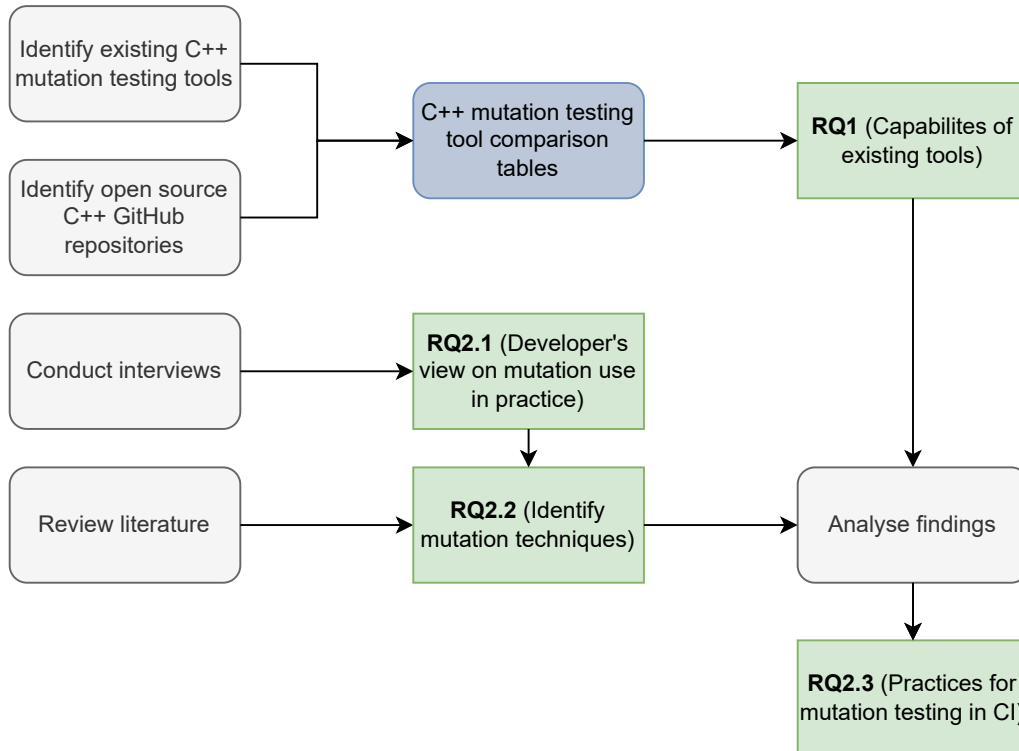
For **RQ1** we compare the features of the tools and evaluate whether the tools could easily be integrated into a development process in the form of a continuous integration pipeline. **RQ2.1** looked into what effective use of mutation testing would entail for a developer. We identified what goals they believed mutation testing could help them achieve. For example, mutation testing may help identify undertested areas of the source code, identify trends in testing over time, or meet other testing goals. **RQ2.2** investigated how we could increase the efficiency of mutation testing by identifying techniques from the literature and features from the mutation tools that can be applied to meet the goals of developers effectively. **RQ2.3** used the data collected in **RQ2.1** and **RQ2.2** to identify guidelines for how mutation testing can generally be applied within an CI context. Figure 4.1 gives an overview of the case study process.

To address these questions, we have:

- (1) **Evaluated C++ mutation tools:** We have evaluated a set of existing mutation tools for the C++ language (Section 4.3). Some tools may support different mutation operators or handle mutation generation and test execution differently. Therefore, we compared the functionality of the tools, and benchmarked both the generation of mutants and the test execution. We also built a proof of concept integration of C++ mutation tools into a CI pipeline.

(2) **Gathered data from stakeholders:** We have conducted semi-structured interviews with developers and product owners at the partner company (Section 4.4.1). These interviews were conducted to understand how stakeholders thought mutation testing could help them effectively in their practice.

(3) **Gathered mutation techniques and practices:** We performed a literature review to find techniques and practices presented in other academic works to identify techniques and practices that could effectively meet the developers' goals (Section 4.4.2).



**Figure 4.1:** Overview of the case study process. The different shades represent different parts of our case study, such as activities (light), deliverables (darker), and answers to our RQ (medium).

## 4.1 Research Design

The research design for the thesis is a case study. What is being studied are best practices when integrating mutation testing in an industrial setting at the partner company of the thesis, Zenseact. A core aspect of a case study is that the conclusions and analysis are limited to a specific context: the employees with mutation testing experience at the partner company. Nevertheless, we will attempt to draw conclusions that generalize, at least, to similarly sized companies in similar problem domains. Since the data collection and analysis were limited to the partner company, that allowed for an exploratory investigation of how mutation testing can be applied in practice, allowing an in-depth analysis of the challenges involving the application of mutation testing into CI. Table 4.1 gives an overview of the case study plan.



**Table 4.1:** Case study plan according to Robson [41] as contained in [42].

<b>Objective:</b>	Explore mutation testing within continuous integration in an industrial setting
<b>Case:</b>	Integration of C++ mutation testing tools into continuous integration pipelines
<b>Theory:</b>	Mutation Testing and Continuous Integration
<b>Research questions:</b>	RQ1, RQ2 (RQ2.1, RQ2.2, RQ2.3)
<b>Methods:</b>	Interviews, literature review and tool evaluation
<b>Selection strategy:</b>	Open-source C++ projects and interview study participants

## 4.2 Context from the Partner Company

Zenseact is a company in the automotive industry with around 600 employees [46] and is owned by Volvo Cars. They focus on the development of autonomous driving and active safety software. The focus of this study is on the employees with mutation testing experience.

Zenseact uses CI to build code and execute test cases. When new code is pushed to the company repository, a process is started that involves both automatic and manual actions. The automated actions are called jobs and are divided into sequential stages; the more expensive a job, the later it is run in the sequence. If any job within a stage fails, the next stage is not started. There are automatic build-and test-avoidance- jobs that are skipped if the affected code has not changed since the last execution of the job. The results from the jobs are then reported back to relevant parties. Failure in the early stages, called pre-merge jobs, will block a merge of the new code, and a manual code review is always needed before the actual merge can be done. Later stages are run after the merge has been done to report any problems caused; these stages consist of the most expensive jobs and are called post-merge jobs. Most of them are run during the day, but some are run at night to avoid blocking other jobs.

Test quality is not assessed automatically; instead, a manual code review is applied to determine the test quality. Test coverage is measured to ensure that test cases execute the code. Mutation could be used at different points in this pipeline, with varying results. Other jobs could help the identification of ineffective mutants in the pipeline. For example, code that does not pass static analysis jobs would be an unproductive mutant, as such a mutant would have been detected before the conclusion of the first stage.

### 4.3 Evaluation of C++ Mutation Tools (RQ1)

To understand the functionality and limitations of the existing mutation testing tools for C++, the tools were installed on the same system and applied to the same C++ projects. We performed benchmarks to collect quantitative data, for example, the number of mutants generated, time spent generating the mutants, time spent executing the mutation tests, and mutation score.

Five existing C++ mutation testing tools were identified by reviewing the literature and searching the Web for tools not mentioned in the academic literature. Table 4.2 shows a list of tools and versions used. MuCPP does not have a version number or changelog, so we cannot specify which version of the tool was used.

**Table 4.2:** C++ mutation tools with versions used for the thesis. MuCPP does not have a listed version on its website. It does, however, have a “last updated” post, but it is unclear if that is when the website was updated or the tool.

Name	Version	Updated	Used with Docker	Source
Dextool	4.1.0-4-g2b5bc097	25 Mar 2022		[19]
Mull	0.17.1	17 Mar 2022		[10]
MuCPP	N/A	7 Jan 2021	X	[7]
Mutate++	bb341d7 (commit)	25 Nov 2020		[31]
CCmutator	66eca5c (commit)	27 Sep 2013	X	[21]

The computer used to perform the benchmarks had an Intel Core i9-10885H processor, 32GB of DDR4 RAM, and a Ubuntu 20.04.3 operating system. MuCPP and CCmutator were installed on a Docker v21.10.12 container using the Ubuntu 20.04 image. They were installed in a Docker container because they had incompatible dependencies of the Clang version with the rest of the tools. As mentioned in Section 2.3, Docker has a negligible overhead on I/O operations and CPU performance but a noticeable impact on network performance. Since we are not dealing with network performance in our benchmarks, we choose not to rerun all tests for our other tools in Docker containers. The settings used for each tool can be found on GitHub [33].

Six suitable C++ projects of varying sizes were chosen on which to apply the mutation tools. We limited ourselves to six projects due to time constraints, as mutation testing can take quite some time to perform. The projects were found by looking at projects used in other academic papers [47, 7] and a list of popular C++ repositories on GitHub. For a project to be selected, it had to have been updated within the last two years, able to compile on the computer used for benchmarking, and had to have a unit test suite. Table 4.3 gives an overview of the projects used, including the number of C/C++ lines of code (LOC), C/C++ files, commit version, and stars given on GitHub to gauge popularity. The number of stars has been rounded to a higher hundred value for consistency since GitHub rounds the number up and does not display the exact number after 10000 stars.

**Table 4.3:** C++ projects with size, GitHub stars, and commit used for the thesis, only C and C++ files counted.

Name	LOC (C++/C)	Files	Commit	Updated	Stars	Source
TinyXML-2	5581	3	a977397	4 Sep 2021	4000	[23]
JSON	60366	104	eec79d4	30 Jan 2022	29300	[30]
Corrade	15359	141	3643585	30 Jan 2022	400	[28]
FMT	42229	65	afbcf1e8	8 Feb 2022	13900	[13]
TimSort	1584	8	e782512	30 Jan 2022	300	[44]
yaml-cpp	18210	54	edadfec	17 Feb 2022	3200	[18]

The documentation of the different mutation tools was compared to identify features and supported mutation operators. The tools were also installed on the same system and applied to the projects listed in Table 4.3, to compare how many mutants they could generate for each project and benchmark how long the test execution took. The documentation was sometimes lacking and not always up-to-date with the current version of the tool, so additional functionality was identified by using the tools. For example, Mull had just undergone a rework in its test execution phase, deprecating the old module performing the test execution and replacing it with a new module that used new conditional flags. However, the tutorials used the old deprecated module and its no-longer-relevant conditional flags. Another example of lacking documentation was in the config files for both Mull and Dextool. Mull did not mention its *includePaths* option to specify what paths to include for the mutation testing. Similarly, Dextool’s config documentation did not mention some of its available options and instead opted to explain them directly in the config file once initialized.

The collected data was then consolidated into tables:

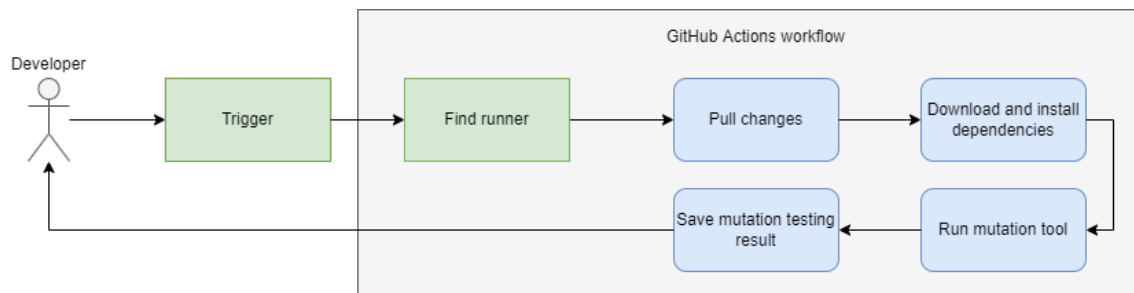
- Table 5.1 that shows if we could install the tool and perform the mutations.
- Table 5.2 compares the functionality relevant to CI.
- Table 5.3 compares the supported traditional mutation operators.
- Table 5.4 compares the number of mutants generated for each project.
- Table 5.5 compares the time it took to generate the mutants for each project.
- Table 5.6 compares the time it took to execute the mutation test using Dextool and Mull.
- Table 5.7 compares the versions of Clang supported by each dependent mutation tool.
- Table A.1 lists class level mutation operators supported by MuCPP.

- Table A.2 lists multi-threaded mutation operators supported by CCMutator.

Using the data collected from the tool testing, we could see that the features and functionality of the tools differed, but also how fast the tools were at different tasks. This gave us an overview of the capabilities of the tools, painting a picture of what existing tools could be used to implement mutation testing and where there was room to improve tools or create new tools to complement the existing tools to fulfill the stakeholder's needs.

As a proof-of-concept for the use of mutation testing for C++ in CI, two CI workflows for mutation testing were created for the TinyXML-2 project using GitHub Actions. The workflows can be found on GitHub [34]. One workflow is triggered by pushes to the repository and only mutates the git diff. The second workflow is triggered periodically, mutating the entire project using the mutation tool(s).

The workflows operate largely the same way; an overview of them can be seen in Figure 4.2. After the workflows have been triggered, GitHub Actions that manages the workflow finds a suitable runner; a runner is an application that will run the jobs in the workflow. The worker then pulls the relevant commit from the repository and then downloads and installs the dependencies along with the mutation tool. After the dependencies have been installed, the mutation tool(s) are used to mutation test the entire project or the git diff caused by the triggering commit, depending on the workflow. The result of the mutation testing is then stored so that relevant parties can inspect the result.



**Figure 4.2:** Overview of the mutation testing CI pipeline using GitHub actions. Rectangles represent actions performed by the GitHub back-end app, and rounded rectangles represent actions performed by the worker.

## 4.4 Usage of Tools in CI (RQ2)

### 4.4.1 Developers' views of effective mutation testing (RQ2.1)

To understand why practitioners have not widely adopted mutation testing and how best to use mutation testing within continuous integration, interviews were conducted with stakeholders in the company to collect qualitative data to analyze.

**Stakeholder interviews:** We conducted semi-structured interviews with seven stakeholders within the partner company; the stakeholders were a mix of software

developers, architects, and managers. The purpose of the interviews was to understand how stakeholders thought mutation testing could best help them in their practice. The semi-structured format allowed stakeholders to present new ideas during the interview. They had varying experiences with mutation testing, ranging from non-existent to active use. An overview of the participants is found in Table 4.4. Something to note in the table is that Zenseact as a company was established in 2020 due to the splitting of the company Zenuity. Any prior time at the company before the creation of Zenseact listed in the table should be attributed to time working at Zenuity.

**Table 4.4:** Interview study participants.

ID	Role	Time at the company	Mutation experience
Participant 1	Software developer	4 years 9 months	Using it actively at work
Participant 2	Software developer	1 year 6 months	Read about it
Participant 3	Cloud architect	6 months	Academic experience
Participant 4	SCRUM master and software developer	1 year 11 months	Read about it
Participant 5	Software developer	4 year 9 months	Read about it
Participant 6	System architect	2 years 4 months	Read about it
Participant 7	Chief safety manager	4 years 10 months	Seen others do it in practice

Before the interview started, the interviewee was asked to consent to the interview being recorded. A short introduction to the basics of mutation testing was also offered before each interview to ensure that all participants had a basic understanding of the topic. We tried to identify how they thought mutation testing could be effectively used in practice in the interview. They were asked, among other things, if they thought mutation testing could improve the testing process, where in the development process they thought mutation testing could be integrated, if they saw value in making mutations a part of the code review process if mutation score could be used as a threshold to deny commits under a certain percentage, the usefulness of mutation score data over time, adverse effects of mutation testing, how much time they think it would be worth spending integrating mutation testing, and how much time they think it would be worth spending analyzing mutation results. If the participant had any prior knowledge about applying mutation testing in their practice, we also asked them where they used it and their opinion on the implementation. See Appendix C for the interview guide.

After the interview sessions, thematic analysis was used to analyze the qualitative data from the interviews. Thematic analysis interprets patterns of meaning within the qualitative data. Meaningful or expressive pieces of data are first identified from the data, called codes. The codes are then given labels representing a summary topic for a set of codes. Once the codes have been identified, they are organized into themes to find patterns of meaning.

The content of the interview was first transcribed and then coded and labeled. To ensure the reliability of the analysis, the two supervisors each performed an

independent coding of a small portion of the interview data (e.g., 10%). Each supervisor used a different part of the interview data. We then compared our findings and assessed the percentage of codes that overlapped, and then differences in coding were discussed. Once consensus was achieved, the author of the thesis individually coded the remaining transcriptions. The codes were then labeled and clustered into themes on a thematic map.

To assign a theme to a code, we first identified the purpose of the code. For example, in the case of the code “*set a particular percentage as a threshold below which change would not be acceptable*”, the interviewee referred to the use of mutation score as an acceptance criterion, so the code was assigned the sub-theme *Mutation score as an acceptance criterion*, which in turn has the main theme *Test quality*. The resulting themes and a theme map are presented along with the results of our analysis in Chapter 5.

### 4.4.2 Techniques to effectively meet the goals of the developers (RQ2.2)

**Literature Review:** We conducted a literature review to examine observations made by other studies on the application of mutation testing in practice—particularly in a CI/CD context—and to identify the techniques that have been presented in the academic literature to perform mutation testing effectively.

The literature review was conducted by searching the scientific database IEEE Xplore [16] for mutation testing articles and filtering relevant ones. Backward snowballing was also used to find relevant papers not found in the database search. The same filtering process was repeated for the citations in the papers. The filtering process included first screening the article’s title to see if it seemed to be pertinent to our research questions. For a title to be deemed relevant, it must be connected to the mutation testing topic. The abstract was further screened if the title passed the screening to see if the article seemed appropriate. For an abstract to be deemed relevant, it had to connect to using mutation testing in practice, optimizing mutation testing, mutation testing in CI or a mutation testing tool. After a paper was screened and deemed relevant, the article’s content was coded and labeled.

The techniques from the literature were then matched with the features offered by the tools evaluated from **RQ1** to generate a combined list of techniques for effective mutation testing.

### 4.4.3 Guidelines for Mutation Testing Within CI (RQ2.3)

We identified guidelines for mutation testing within CI from the combined results of **RQ1**, **RQ2.1** and **RQ2.2**. The guidelines are based on the developers’ goals at the partner company but can still be informative for other companies and developers with a similar context.

**Match techniques and practices with themes:** The techniques and practices

**Table 4.5:** Keywords used for literature review database searches.

Search keywords
Mutation testing
Mutation testing C++
Mutation testing continuous integration
Mutation testing practice
Mutation testing industry
Mutation testing at scale
Mutation testing case study
Mutation testing cost reduction
Mutation testing LLVM

for mutation testing identified in **RQ2.2** were matched with the sub-themes generated in **RQ2.1** to see what techniques and practices could be used to help achieve the developers' goals for effective mutant testing. This happened naturally as a part of an inductive process of the thematic map. Topics from **RQ2.2** were connected with similar sub-themes. For example, the topics *timeout tests* and *selective mutation selection* were matched with the sub-theme *time-aware feedback*. The resulting connections can be seen on a map in Chapter 5.





# 5

## Results

### 5.1 Evaluation of C++ Mutation Tools (RQ1)

Different mutation testing tools exist, each with its advantages and disadvantages. This study looked at five tools for mutating C++ code, Dextool, Mull, MuCPP, Mutate++, and CCmutator. See Chapter 2 for more info about the tools.

#### 5.1.1 Installing and using the tools

We started by trying to install and run the different tools. We then tried to generate mutants and run mutation testing using the tools if that was successful. MuCPP and CCmutator had to be installed in Ubuntu Docker containers due to incompatible Clang version dependencies with the rest of the tools.

The result of these activities can be found in Table 5.1. However, something that is not reflected in the table is that three<sup>1</sup> of the forty-three mutation operators for Mull caused an error when used<sup>2</sup>. But with them turned off, there was no issue generating mutants and executing the mutation testing using Mull. There were some problems installing Mutate++, as it had incorrect requirements file with incompatible Python library versions. After changing to newer versions of the libraries than the ones specified, we were able to find library versions that were compatible with each other. We were then able to run the tool and generate mutants. However, we were not able to execute the mutation tests due to the Mutate++ raising an error<sup>3</sup>.

We were also unable to install CCmutator using its installation instructions. However, after installing wget and python2 in our Docker environment, we were able to install the tool using the provided installation script. Nevertheless, when we tried to generate mutants using a CLI command<sup>4</sup>, we got an error<sup>5</sup>. We choose not to

---

<sup>1</sup>Mull's mutation operators `cxx_logical`, `scalar_value_mutator` and `negate_mutator` caused issues.

<sup>2</sup>"[error] Uh oh! Mull corrupted LLVM module."

<sup>3</sup>Command 'patch -p1 --input=/tmp/tmpqu6cu8aq (...) /tinyxml2/tinyxml2.cpp' returned non-zero exit status 1"

<sup>4</sup>"(...) /llvm-3.2.src/install/bin/opt -basicaa -debug -load ./mutate\_Mutex.so -Mutex -analyze [...] /tinyxml2.cpp.o"

<sup>5</sup>"(...) /llvm-3.2.src/install/bin/opt: [...] /tinyxml2.cpp.o:1:1: error: expected top-level entity"

spend more time trying to fix the errors caused by Mutate++ and CCmutator.

**Table 5.1:** Summary of the activities and the corresponding tools in which these activities were successful.

Activity	Dextool	Mull	MuCPP	Mutate++	CCmutator
Could compile or start the tool.	X	X	X	X	X
Could install or compile the tool by following the installation guide.	X	X	X		
Were able to generate mutants without modifying the target projects, except for compile commands.	X	X	X	X	
Executed mutation testing using the tool.	X	X	X		

**RQ1 (Evaluation of C++ Mutation tools):** Dextool and MuCPP installed and performed mutation testing without issues. Mull had some mutation operators that make the tool crash but works fine otherwise. Mutate++ and CCmutator did not work due to throwing errors.

### 5.1.2 Comparing the feature set of the tools

Each of the tool’s features affect how the tool can be utilized, how well the tool can be integrated into a CI environment, or what other tools might be needed to complement the tool to be used within a CI environment. For example, a tool might be able to generate mutants but not execute test cases and calculate the mutation score, in which case, another tool would be needed to execute test cases and calculate the mutation score. Furthermore, another essential feature is that the tool should not rely on a GUI (Graphical User Interface) and instead offer a CLI (Command Line Interface) or another way to use the tool programmatically since this is crucial for CI integration. We looked at what tools had been updated in the last two years to see if any of the tools seemed abandoned.

As we can see in Table 5.2, the features of the different tools differed. CCmutator was the only tool seemingly abandoned; it had not been updated within the last two years, making its future support questionable. All tools except CCmutator were able to execute mutation testing; another tool is needed to test the mutants generated by that tool. While Dextool and Mull can mutate based on a git diff according to their respective documentation, only Dextool could do it. The functionality seems to have been removed from Mull in a recent update.

**RQ1 (Evaluation of C++ Mutation tools):** Dextool has support for the most features relevant to CI integration and is the only tool that could mutate a git diff. Mull had support for fewer features but may still be appropriate for use. MuCPP cannot skip mutants that are not covered by tests, potentially making it waste time on mutants that the test suite can not kill. Mutate++ relies on a GUI, making it inadequate for CI integration. CCmutator needs another tool to execute test cases on its mutants.

**Table 5.2:** Features relevant to CI integration by the different tools.

Feature	Dextool	Mull	MuCPP	Mutate++	CCmutator
Command line interface (CLI).	X	X	X		X
Generate mutants	X	X	X	X	X
Execute mutation testing	X	X	X	X	
Limit the mutation operators to be used by to a subset specified by the user.	X	X	X		X
Specify what lines of code to mutate.				X	X
Mutate the Git diff, changes made to the code compared to another state of the code.	X				
Only perform mutation testing on lines of code that the test suite has covered.	X	X			
Stop executing mutation tests after a specific number of mutants have been detected as live.	X				
Skip already killed mutants if no relevant code to the mutant has been changed.	X				
The generated mutants can be used outside the tool.	X	X	X		X
Detect redundant unit test cases that do not uniquely kill any mutants or tests that kill the same mutants.	X				
Resume mutation testing after an interruption.	X			X	
Supports mutation schema, injecting multiple mutants in toggle able states.	X	X			
Supports parallelization of the mutation testing phases.	X	X			
Flag mutants to be ignored for future runs	X			X	
Timeout mutation tests if they have run for a specific time.	X	X	X	X	
Project has been updated within the last 2 years.	X	X	X	X	

### 5.1.3 Comparing the mutation operators of the tools

Our tools under evaluation offered different mutation operators identified by looking at the documentation for the tools. A table with examples for all the supported mutation operators can be found in Appendix A.3. Mull did not follow the conventional naming scheme for its mutation operators, so we mapped its operators against the conventional operator names; see Appendix B for this mapping.

As shown in Table 5.3, Dextool, Mull, MuCPP, and Mutate++ all support some of the traditional mutation operators. The only tool not supporting these mutation operators is CCmutator. Instead, the CCmutator tool is focused on multi-threaded C++ programs [21] and has support for 38 multi-threaded mutation operators; see Appendix A.2 for a list of them. MuCPP, on the other hand, on top of supporting traditional mutation operators, as mentioned before, also has support for 30 language-specific class-level mutation operators; see Appendix A.1 for a list of these. This makes MuCPP the tool with the most variety of supported operators.

**Table 5.3:** Traditional mutation operators that are supported by the mutation tools.

Operator	Name	Dextool	Mull	MuCPP	Mutate++	CCmutator
AOR	Assignment operator replacement		X	X		
AOR	Arithmetic operator replacement	X	X	X	X	
DCR	Decision/ condition requirement	X				
ROR	Relational operator replacement	X	X	X	X	
SDL	Statement deletion	X	X	X	X	X
ABS	Absolute value insertion					
COR	Conditional operator replacement	X	X	X	X	
LCR	Logical connector (operator) replacement	X	X	X		
UOI	Unary operator insertion	X	X			
CR	Constant replacement	X	X		X	
SVR	Scalar variable replacement		X			
SBR	Statement block removal	X				
AOI	Arithmetic operator insertion		X			
ADS	Arithmetic operator deletion			X		
COD	Conditional operator deletion			X		
COI	Conditional operator insertion			X		

**RQ1 (Evaluation of C++ Mutation tools):** MuCPP has support for the most mutation operators, including 11 traditional and 30 language-specific operators. CCmutator does not have support for traditional operators but is the only tool to offer multi-threaded mutation operators. The other three tools only support a small number of traditional operators.

### 5.1.4 Mutation generation results

The mutation testing tools vary in what mutation operators they support and how they generate their mutants. See Table 5.4 for the resulting mutation amount and Table 5.5 for the mutation generation speed on the different projects. MuCPP requires a Makefile to function; if there was none in the project, we did not create one, so the tool was not run on these projects. Mull has a bug where, if a template function is present, Mull might not correctly report the total number of mutants. Hence, the number of mutants for these projects cannot be trusted. We only ran the mutation generating benchmarks for the tools that we could perform the mutation generating with; see Section 5.1.1 for more details. Dextool requires the project to be compiled before the tool is used to generate the mutants, so we added the compile time for each project to the time spent generating mutants in the result.

**Table 5.4:** The number of mutants generated per project and per tool. Mull might not report the correct total mutation amount if template functions are present. Projects with templates that might affect the mutation amount are marked with a \*. Projects marked with — for MuCPP did not have the prerequisite Makefile.

Project	LOC (C++/C)	Dextool	Mull	MuCPP
TinyXML2	5581	1907	698	3158
JSON	60366	6407	528*	9266
Corrade	15359	9573	2314	—
FMT	42229	9815	1998*	—
TimSort	1584	1073	279	—
yaml-cpp	18210	6235	2147	—

**Table 5.5:** Time (in seconds) spent generating mutants for the projects using the mutation testing tools. Projects marked with — for MuCPP did not have the prerequisite Makefile.

Project	LOC (C++/C)	Compile time	Dextool	Mull	MuCPP
TinyXML2	5581	2.0s	3.7s	2.1s	59.0s
JSON	60366	34.9s	44.9s	576.7s	542.8s
Corrade	15359	13.1s	39.4s	72.0s	—
FMT	42229	47.2s	65.9s	1033.0s	—
TimSort	1584	22.5s	36.4s	94.8s	—
yaml-cpp	18210	15.6s	29.9s	62.8s	—

**RQ1 (Evaluation of C++ Mutation tools):** MuCPP generated the most mutants but was slow to do so, and it could not generate mutants for all projects. Dextool generated the second most mutants and was generally the fastest tool for generating mutants. Mull generated the least mutants and took the longest to do so, but this will likely be made up for in a later mutation test phase.

### 5.1.5 Mutation execution results

Dextool and Mull had built-in support to perform mutation testing. See Table 5.6 for the time spent on mutation execution and the resulting mutation score. Due to the limitation of Mull only accepting one test binary, a test binary is a compiled binary file that executes a set of tests; only one test binary was provided to the other tools as well for the testing phase. This means that many mutants are skipped because the given test do not cover them.

As can be seen by comparing Table 5.6, Mull is significantly faster in some scenar-

ios, as it does not have to recompile the entire project between test runs. In the case of the TimSort project, Mull executed mutation testing for its 279 mutants in 2.25 seconds, while Dextool took more than 1 hour to execute its 1073 mutants. That’s 123.78 mutants executed per second with Mull and 0.017 mutants for Dextool. Something to keep in mind when looking at these tables is that most of the test binaries were only part of the projects’ tests, making the test execution phase complete more quickly than usual, while the compilation time stayed the same. This potentially skews the result slightly. TinyXML2 and Yaml-cpp had a single test binary that included all the tests. Here, we can see that Mull also performs faster than Dextool, but not to the same magnitude as in the other project benchmarks.

**Table 5.6:** Mutation execution benchmark results for each project using Dextool and Mull. Only one test binary could be executed at a time due to limitations in Mull. The baseline column represents the time it took the test suite to complete on its own, without any tool involved.

Project	Test Suite	Operators	Baseline	Dextool			Mull		
				Time	Mutants	Score	Time	Mutants	Score
TinyXML2	xmltest	All	0.1s	2146.1s	1907	77.00%	368.4s	698	82.00%
JSON	unit-algorithms	All	0.1s	12980.1s	6047	2.32%	10.1s	528	9.00%
Corrade	MainTest	All	0.1s	47009.9s	9573	2.35%	1.5s	2314	5.23%
FMT	core-test	All	0.1s	66364.6s	9815	2.30%	1.0s	1998	1.80%
TimSort	cxx_98_tests	All	0.4s	5398.9s	1073	59.30%	2.3s	279	1.43%
yaml-cpp	yaml-cpp-tests	All	0.8s	29208.0s	6235	72.90%	7254.5s	2147	75.41%

**RQ1 (Evaluation of C++ Mutation tools):** Mull was the fastest tool in the execution of the mutation tests but was limited to using only one test binary at a time. Dextool was slower but supported multiple test binaries simultaneously, making it easier to adapt to projects that might not compile all tests into one binary.

### 5.1.6 Clang dependency

The different mutation tools have dependencies that should be considered when identifying a tool to work with, as some tools might be incompatible with the toolchain in place. A notable dependency is Clang, a front-end compiler for C++ and other programming languages. Four of the five mutation tools examined relied on different versions of Clang; the only one not dependent on Clang is Mutate++. We could not integrate any of the mutation tools examined in this project with the toolchain at the partner company Zenseact since they use a newer Clang version incompatible with the mutation tools. See Table 5.7 for the Clang versions supported by each tool.

### 5.1.7 CI workflow implementation

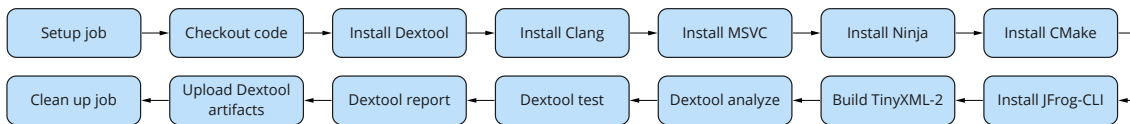
We created two CI workflows for the TinyXML-2 project using GitHub Actions as proofs-of-concept for the use of mutation testing for C++ in CI. The first workflow is

**Table 5.7:** Versions of Clang supported by each dependent mutation tool.

Tool	3.2	3.6	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0	12.0	13.0	14.0
MuCPP		X							X				
Mull								X	X	X	X	X	
Dextool			X	X	X	X	X		X	X	X	X	
CCmutator	X												

triggered on a schedule and uses Dextool or Mull, depending on the configuration, to perform the mutation testing on the entire project. The other workflow is triggered by pushes to the repository and uses Dextool to mutate the git diff introduced by the push. The result of the mutation testing is stored for later review. An overview of the GitHub Actions workflow can be seen in Chapter 4 in Figure 4.2.

The workflows are run on runners using the GitHub Docker runner image based on Ubuntu 20.04. The scheduled workflow consists of two jobs; a job for Dextool and another job for Mull. The git diff workflow only has a Dextool job. The job is divided into steps; see Figure 5.1 for the steps in a Dextool job. The resulting artifact is then uploaded to Artifactory. In the case of Dextool, this is an SQL file that can be used to generate a mutation test report. An example report can be seen in Figure 5.2.

**Figure 5.1:** Overview of the steps for the Dextool CI job.

**RQ1 (Evaluation of C++ Mutation tools):** These workflows demonstrate that both Dextool and Mull can be applied in CI as part of workflows. Our proof of concept is available as an open-source repository [34].

## 5.2 Mutation Testing Within CI (RQ2)

### 5.2.1 Developers' Views of Effective Mutation Testing (RQ2.1)

This section will present and discuss the themes generated by analyzing the qualitative data from the interviews. A list of interviewees can be found in Section 4.4.1. The purpose of the interviews was to obtain the developers' view of effective mutation testing. A list of the generated themes with descriptions can be found in Table 5.8 and an overview of the generated sub-themes in Figure 5.3, see Appendix D.1 for a description of the generated sub-themes.

Overview

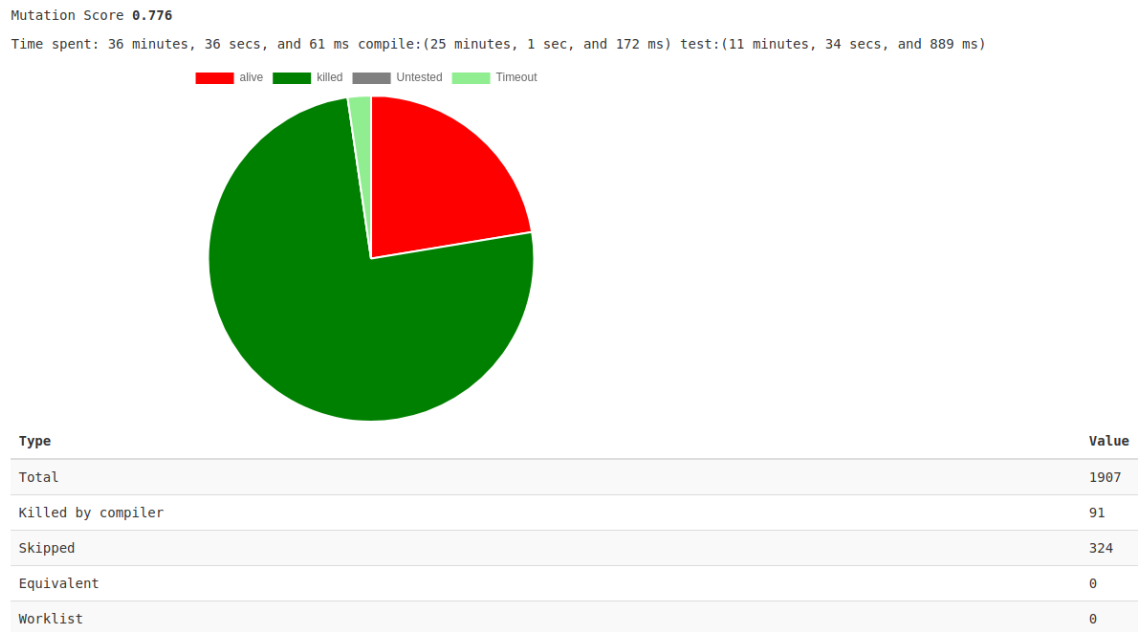


Figure 5.2: Dextool mutation report example.

Table 5.8: Description of generated themes from the interviews.

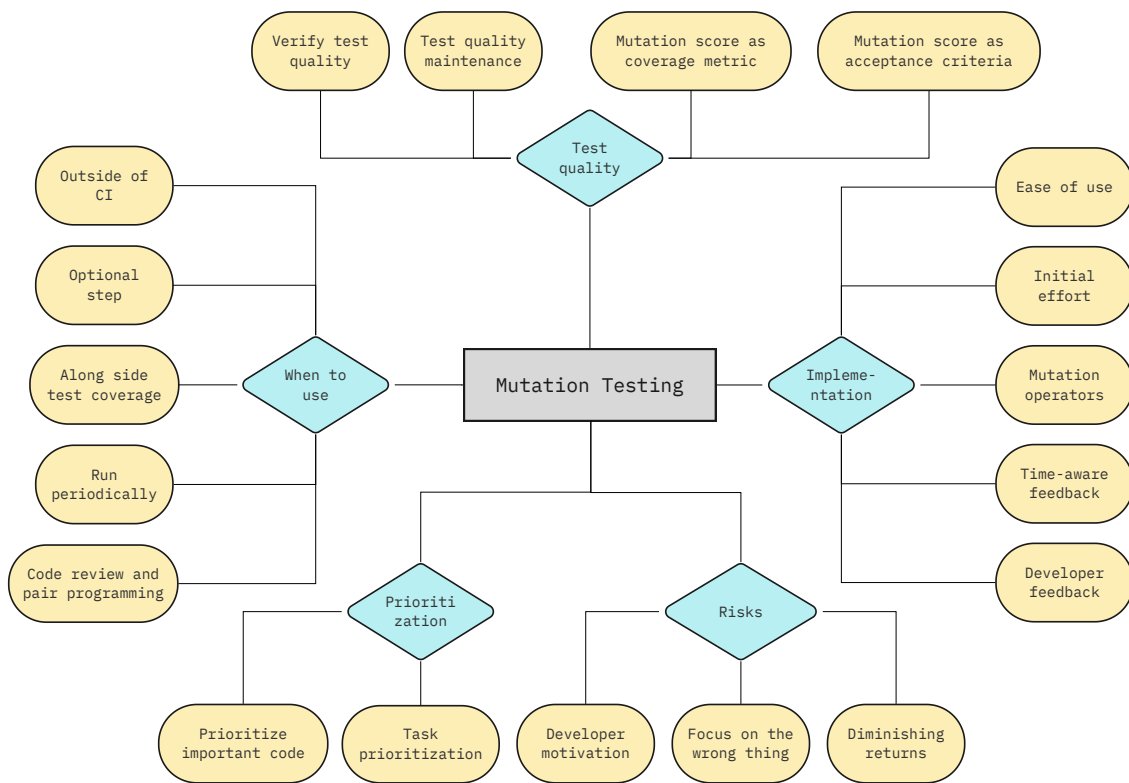
Theme	Description
Test quality	How mutation testing can affect the project’s test quality by indicating areas of the code that need more testing or identifying missed edge cases.
Prioritization	What areas of the code to prioritize for mutation testing or times when another task might be too important to be slowed down by mutation testing.
Risks	Risks that might affect how effective the mutating testing is. Such as diminishing returns from overapplying the technique or inexperience with the technique leading to a bad interpretation of results.
Implementation	Aims and technical details to consider when integrating mutation testing into a workflow.
When to use	When, and where to apply mutation testing in practice as a part of the development and test maintenance process.

5.2.1.1 Test Quality

Mutation tests aim to judge the quality of the test suite. This section will discuss how developers think mutation testing can affect test quality.

**Verify Test Quality:** Most of the participants reported that they see mutation





**Figure 5.3:** Overview of generated themes from the interviews. A rectangle represents the topic, a rhombus represents themes, and an oval represents sub-themes.

testing in CI as a way to verify test quality. When developing a test suite, mutation score can be used to indicate the overall quality of the test, and live mutants can be used to identify edge cases that could have been missed. The mutation score can also be used to argue that not only has the code been tested, but that the quality of the tests is this good.

“Good sanity check, are my tests actually covering everything.” - Participant 1

“Make our case stronger, like we don’t just have tests, we can prove that our tests are this good.” - Participant 4

**Test Quality Maintenance:** When modifying code, tests that test the changed code should sometimes also be updated to reflect the new behavior; for example, if the interface has been changed. The concept of maintaining tests and keeping them up-to-date is called test maintenance. Participants suggested using mutation testing to verify that the test suite’s quality is trending in a positive direction when performing test maintenance by looking at how the mutation score has changed over time. This is especially important for delivered code, where faults not caught by the test suite could impact the customer.

“Trend over time could also be important to basically say that it would be an indication that you are developing your test suite toward a better state.” - Participant 2

“If it’s code which is delivered, then it’s really important to continue keeping the score trend going down.” - Participant 5

It is not just changed code that could benefit from mutation testing. The surviving mutants for existing test suites could be seen as technical debt that needs to be fixed to improve the quality of the test suites.

“[Live mutants can be] treated as technical debt, your tests are of bad quality and therefore, it needs to be fixed.” - Participant 4

**Mutation Score as a Coverage Metric:** Similar to how line coverage can be used to measure which lines of code are executed by tests, mutation score can be calculated for the different parts of the code to indicate the quality of the tests covering the code. A developer can then investigate the parts of the code indicated to have lower test quality to assess if the mutants there are worth killing.

“I think the mutation score is some good metric, but of course the usefulness, you know the accuracy, whether the number of mutants are actual problems is where the real value of this is, not just the score by itself.” - Participant 4

“Good to have the score, but I think it’s even more important to find the cases you are missing.” - Participant 1

**Mutation Score as an Acceptance Criterion:** Like structural coverage metrics, the mutation score can be used as an acceptance criterion for new changes in the repository. Code changes below a set threshold could be flat-out rejected or have the developer motivate why the threshold was not met. The threshold could be updated if the mutation score changes; this could be due to more mutants being killed or a change in the mutation operators used to generate the mutants. Having this threshold could encourage the developer to maintain the test suite’s quality.

“Set that threshold in the test, and then you have to motivate in a code review if you raise that.” - Participant 5

“If you change some code, run the mutation tests on it, and if the score is now better, either because you made some changes or you fine-tuned the operators. I think you should definitely update the threshold.” - Participant 4

But a threshold is not always ideal; for example, enforcing a threshold for code

under development could make the developer spend time writing extensive tests for code that might not even be in the final product.

“[When] Introducing new code which is not enabled yet, and that might be okay to during some part of the development, it’s okay that it rises because we want it to be a full feature before we can start getting the quality, work on the quality.” - Participant 5

**Prioritize Important Code:** When applying mutation testing on a large code-base, there could be many living mutants, making it difficult to know which parts of the code to prioritize. Therefore, it might be beneficial to focus on important parts of the code first and only run mutation testing on this code. The developer could then use the result to identify the weakest parts of the important code and prioritize these.

“I would like to start with the components which are most important, and maybe get an overview first, and then see, okay, where do I have the least score and start working there, and put effort into those.” - Participant 1

### 5.2.1.2 Developer Impact

An important aspect to keep in mind when introducing a new tool or changing an old tool in a workflow is how it will affect people who will use it in their day-to-day work. This section will look at how developers perceive the impact of mutation testing in their tasks.

**Developer Motivation:** If the tool provides false positives or does not work, it will cause an interruption in the developers’ workflow, potentially making them frustrated. Furthermore, results that are not interesting would cause the developer to be discouraged from looking into the other results. In contrast, if the results were interesting, the developer would feel encouraged to look into the other results. Therefore, for developers to be motivated to use mutation testing, it should work seamlessly and produce productive results. Otherwise, the developer might lose faith in the technique.

“If you get false positives, if the tooling is not working or if the code you write is not compatible with it somehow, so you get noise in your workflow.” - Participant 5

“If I would get failures that are typically not interesting, I would feel very discouraged to look more into the failures, but if they were actually very on spot, then it would be super interesting.” - Participant 5

Another vital factor to consider to not interfere with the developers’ workflow is that the tool itself should be easy to use once someone has set it up, as to not add additional toll on the developer, which might lead to the developer being demotivated

to use the tool.

“I would assume that for the users of this, after somebody has set it up, then I expect it to be as easy as using the unit testing framework.” - Participant 4

**Focus on the Wrong Thing:** It is up to the developer to interpret the mutation result. There is a risk that the developer focuses on killing the mutants without considering the context, potentially missing a bigger problem in the code.

“If you over apply it and look directly at those [mutation testing] results, and not in the context, it might hurt in that people just add blindly to cover the cases.” - Participant 1

Another potential adverse effect of a developer blindly focusing on killing mutants is that it can cause developers to develop a test suite that is too strict. This could make it challenging to implement alternative ways to achieve the same solution; it could be that another solution can solve the same problem in a better way, but very strict unit tests for the internal workings of the code make it cumbersome to change the code, resulting in the developer feeling demotivated to do it.

“There is not like always one single solution to it. But it could be that other ways are allowed as well, and you don’t want to have a test suite to like narrow everything down.” - Participant 6

Another risk of mutation testing is that the developer is encouraged to perform white-box testing, which is not always the best option, according to Participant 4. White-box testing tests the internal structure of software, and black-box testing does the opposite and tests the functionality. Mutation testing can be considered white-box testing since mutation testing assesses the ability of test suites to detect subtle changes in the code and not the functionality of the software. This could lead to a focus on testing the internal workings of the code in isolation and not the actual combined output, which might not always result in the correct functionality as stated by the requirements for the software.

“Mutation testing, indirectly promotes white-box testing. ... White box testing isn’t always the best choice, or it’s good to have a bit of a mix of an approach of black-box testing and white-box testing.” - Participant 4

**Task Prioritization:** Mutation testing is a slow process, and using the technique might not always be the best use of developers’ time. A developer is typically limited to working around 8 hours per day. They might have more pressing matters to attend to than improving the quality of the already existing test suite. Similarly, it might not be worth using mutation testing to block important tasks, such as deploying a critical bug fix.

“Slow process and we are limited to working like 8 hours each day.” - Participant 3

“Would I want to stop the important bug fix, probably not.” - Participant 1

**Diminishing Returns:** Naturally, the more times mutation testing is used in the same test suite, and code changes are made to kill the live mutants, the fewer mutants might be alive on the next run. Therefore, if killing the most important mutant has been prioritized each time, the value of killing mutants diminishes with each mutant killed. Consequently, it might not be time-efficient to over-apply mutant testing on one part of the code.

“It [mutation testing] has somewhat diminishing returns.” - Participant 1

### 5.2.1.3 Implementation

The implementation of the mutation tool will affect the effectiveness with which developers can use the tool. This section will look at how the interviewees think a mutation tool should function in their day-to-day work.

**Ease of Use:** It should be easy for a developer to use mutation testing in their day-to-day work once it has been integrated. As much as possible should be automated; for example, when additional tests are added, or existing ones are changed, they should automatically be added to the mutation testing. If it is not easy, it might affect their willingness to use the tool, as mentioned in the Developer Motivation sub-theme.

“Should definitely be automated. Ideally, a developer, when writing tests, shouldn’t have to bother with it at all.” - Participant 4

**Initial Effort:** There could be considerable work to get mutation testing to work within the toolchain. The mutation tool(s) would have to be configured to work with the project, with varying efforts depending on the tool. Participant 1, who uses mutation testing with Java, mentioned that it is as simple as adding a code block in Maven, a software project management tool, to enable mutation testing. However, the tools for C++ can require more setup, as mentioned in Section 5.1.

“In Maven there is a specific block that you essentially just add and it runs your JUnit tests.” - Participant 1

“Quite a bit of effort to actually make it work properly, or at least cover the realistic cases.” - Participant 3

Furthermore, developers unfamiliar with mutation testing would have to be trained

appropriately; to understand how mutation testing works to be able to analyze the mutation result better.

“The teams would have to be trained appropriately, and make sure that one is willing to take responsibility.” - Participant 3

**Developer Feedback:** Feedback from the mutation result should show which mutants survived, why they survived, and what parts of the code they changed. It should be easy for the developer to get an overview of where the mutants are located in the code and what type of mutant they are so that the developer can easily analyze the result and act on it accordingly.

“Group the live mutants. I assume there are some nice tools for that. So that even if you have a bad score, and you only care about some mutants that you want to look at, then it would be nice to see how they are related to each other” - Participant 6

**Time-aware Feedback:** Developers do not want to sit around waiting for feedback. If used in CI, then the mutation testing should ideally take as much time as the other jobs in that stage. Even adding a minute of additional waiting time can be cumbersome. It is not just in a CI context that developers want rapid feedback; the same applies when running mutation testing on their machine. The developer wants rapid feedback so that they can work on the problem and then move on.

“If it [mutation testing] were half an hour, well yeah, I wouldn’t be running that.” - Participant 1

“I can leave the thing running for like a few hours, and that’s fine, sometimes. Other times I really want this to be done quickly so that I can move on.” - Participant 3

**Mutation Operators:** A way to speed up mutation testing is by omitting mutation operators, thus reducing the number of mutants. Participant 1, who uses mutation testing in their day-to-day work usually uses all available mutation operators. However, mutation operators have a time-to-value ratio. Generating the most effective mutants utilizing a subset of mutation operators could be an excellent trade-off to save time. Mutation operators could be prioritized based on the mutation operators that usually generate productive mutants.

“Overall, the value for time or time for value ratio. It feels like getting 90% of the effective mutants is a sweet spot. Once you reach that, then the rest is almost a waste.” - Participant 4

“Selecting the ones you usually have mistakes on is, I guess, a prioritization.” - Participant 1

#### 5.2.1.4 When to use

Mutation testing could be applied in multiple stages of development. This section reports on when developers think it can be used effectively.

**Code Review and Pair Programming:** Mutation testing could be used as part of the code review process to judge the quality of the tests. As mentioned in the sub-theme *Mutation Score as a Coverage Metric*, mutation score could be used as a coverage metric to get an indication of the quality of the different parts of the code. There is not always a code review checkpoint; sometimes, pair programming is used where the code is reviewed continuously. However, it still makes sense to use mutation testing after a pair-programming session to check the quality of any created tests and discuss whether any action should be taken.

“Code review checkpoint. There I think it makes sense, and there I think code coverage also makes sense.” - Participant 1

“We are doing pair programming instead. And then the same thing goes there.” - Participant 1

**Run Periodically:** Due to the relatively high machine cost of performing mutation testing, it could potentially take up a lot of machine time when used in CI and delay other jobs, especially if it is done on a large codebase. However, not all jobs have to be run when new code is pushed to the repository, and the more expensive jobs could be run after the code has been merged. Mutation testing could be run as a post-merge job only when there are free computing resources, for example, at night or on weekends when the day-to-day work will not be affected. After the mutation test has been performed, someone could take the responsibility of checking the result. This could be the author of the commit or someone who will send the result to the affected parties.

“We don’t want it to take up too much machine time in CI.” - Participant 2

“Once every few days where we run these tests, and then there is someone who will take responsibility for actually checking this [result] and sending the information to the affected parties.” - Participant 3

**Outside of CI:** Running CI jobs can be slowed by overhead. There could be a limited amount of machine time available, causing a queue of jobs that want to run, and there might be high-priority jobs that get put at the front of the queue. Therefore, it is beneficial to allow developers to run a mutation testing job on their machine, not just in CI. This would also enable the developer to get feedback as soon as they write the test case, without the need to push the code to a repository.

“Run things on CI, there is some kind of overhead, and that can be a lot. So, definitely, you should be able to run it locally.” - Participant 4

“As soon as you have a test case written, then you can start making use of [mutation testing] that.” - Participant 5

**Optional Step:** Mutation testing does not necessarily have to be a forced step; it could be an optional step for developers or teams who want and have time to improve their test quality. It could be offered as an optional tool to use on your machine, as mentioned earlier, or as an optional job to run in CI.

“Optional part of your checking or as git hooks.” - Participant 1

The developer would not necessarily have to use mutation testing regularly; it could be used when new tests are added or to test old tests when there is spare time.

“In a new situation where I’m writing more or different types of tests.” - Participant 1

### 5.2.1.5 Summary

Below, we summarize the key findings from the participants’ answers and our theme analysis.

**RQ2.1 (Effective mutation testing according to developers):** Mutation testing should have minimal impact on the workflow; it should work correctly, be as easy to use as existing tooling, be automated as much as possible, and give feedback as fast as possible without affecting other jobs. The initial focus of mutation testing should be on testing the important parts of the code to get faster feedback on this code area. Furthermore, the mutation score can be used to indicate which parts are in the greatest need of being looked at. Mutation testing can also be used for new code after a programming session to see how the new code affected the test suite’s quality.

### 5.2.2 Techniques for Effective Mutation Testing (RQ2.2)

As with most things, mutation testing must be considered in a time-to-value context. This section will look at the techniques and practices identified for effective mutation testing from the literature review and tool comparison; see Section 5.1 for the tool comparison.



### 5.2.2.1 Techniques for Effective Mutation from the Literature Review

A focus of many techniques in the literature is optimizing mutation testing by reducing the number of mutants generated while still producing valuable mutants. While for the mutation tools, there seems to be more of a focus on increasing the speed without impacting the number of mutants, with, for example, parallelization. The techniques and practices identified in the literature review and tool comparison presented in Table 5.9 will be discussed below:

**Selective mutation selection:** A subset of mutation operators can be selected that generate the most effective mutants to reduce the time spent testing mutations while still keeping the most productive mutants. As mentioned in Chapter 3, Offut et al. [32] have shown that the subset of the mutation operators ABS, AOR, LCR, ROR, and UOI is almost as effective at generating mutants as non-selective mutation testing. However, Petrović et al. [38] have shown that in their use case, ABS generates nonproductive mutants and should be replaced by SBR and note that this may be because of a function of the style and features of their codebase. Therefore, to reduce the time it takes to perform mutation testing while still generating productive mutants, it may be worth experimenting with what operators most often produce productive operators for a specific codebase.

Papers: [38, 45, 8]

Tools: Dextool, Mull, MuCPP, CCmutator

**Mutant schema:** Instead of inserting one mutant at a time, multiple mutants can be inserted at once in inactive states. The mutants can then be activated one at a time without recompiling in between, making it significantly faster to switch between them.

Papers: [27, 45]

Tools: Dextool, Mull

**Stop mutation testing after X iterations:** Normally mutation tools timeout mutation tests after a specific number of seconds have passed. An alternative way to handle this is to stop after a statement has been repeated a specific number of times. This could potentially lead to a faster stop of an infinite loop.

Papers: [27]

Tools: —

**One mutant per line of code:** Killing one mutant often leads to the death of other mutants generated from the same line of code. Therefore, generating more mutants on the same code can have diminishing returns. To reduce the diminishing return, one mutant can be generated per line of code.

Papers: [39]

Tools: —

**Mutation selection based on historical data or machine learning:** When limiting mutation generation to one mutant per line of code or statement, an operator could be selected that most often generates a productive mutant for that code

## 5. Results

**Table 5.9:** Mutation techniques and practices from the literature and mutation tools.

Technique	Description	Paper	Tool
Selective mutation selection	A sub-set of mutation operators are selected that generate the most effective mutants.	[38, 45, 8]	Dextool, Mull, MuCPP, CCmutator
Mutant schema	Multiple mutants are inserted at ones in inactive states.	[27, 45]	Dextool, Mull
Stop mutation testing after X iterations	Time out a test after a specific number of iterations.	[27]	
One mutant per line of code	Only generate one mutant per line of code.	[39]	
Mutation selection based on data	When limiting mutation generation to one per line of code or statement, select the operator most likely to generate a productive mutant based on historical data or machine learning.	[26, 38]	
Only run relevant tests	Only run tests that can kill the mutant, tests with no coverage of the mutated line of code cannot kill it.	[39]	
Mutate code changes	Only mutate changed code.	[26, 39]	Dextool
Filter code based on patterns	Filter code to mutate based on a pattern match.	[38]	
Introduce real world bugs	Create code mutations based on real bugs using machine learning.	[3, 4]	
Present a limited amount of mutants	Only present a limited amount of live mutants to the developer.	[38]	
Only mutate code covered by tests	The tool can identify what lines of code are covered by tests and only mutate those.	[38]	Dextool, Mull
Specify tests to use	Specify what tests to use for the mutation testing.	[38]	Dextool, Mull, Mutate++
Flag mutants	Flag mutant as irrelevant, so that it will be ignored or avoided in future runs. Similarly, useful mutants could be flagged as helpful to build historical data.	[38]	Dextool, Mutate++
Specify code to mutate	The user can specify what lines of code to mutate.		Mutate++, CCmutator
Stop after a number of live mutants	After a specific number of live mutants have been detected, the mutation testing stops.		Dextool
Skip killed mutants	The mutation result is stored between runs, so that killed mutants can be skipped if there has been no change to the test suite.		Dextool
Detect redundant tests	Redundant tests that do not uniquely kill mutants are flagged.		Dextool
Resume mutation testing	Mutation testing can be resumed after an interruption.		Dextool, Mutate++
Timeout tests	Timeout is a set value or can dynamically be set for every test suite based on the normal run time.		Dextool, Mull, Mutate++
Parallel mutation testing	Parallelization of the mutation testing phases.	[38]	Dextool, Mull

structure instead of selecting a random operator. The most effective mutation operator for the code structure could be selected based on historical data or machine learning.

Papers: [26, 38]

Tools: —

**Only run relevant tests:** Run tests that can kill the mutant, tests with no coverage of the mutated line of code cannot kill it and are thus a waste of time to run. This could be limited to selecting test suites to run or individual tests.

Papers: [39]

Tools: —

**Mutate code changes:** When the quality of the test for new code is to be assessed, for example, in a code review, mutation testing could be significantly sped up by testing only the new code and not wasting time on irrelevant old code.

Papers: [26, 39]

Tools: Dextool

**Filter code based on patterns:** Not all code should be mutated. For example, there could be print debug lines that are irrelevant to the code's functionality; mutating these would be a waste of time. The code could be blacklisted based on a pattern match to ignore reoccurring known irrelevant code statements.

Papers: [38]

Tools: —

**Introduce real-world bugs:** A tool could be trained using machine learning to introduce real bugs. The tool could be trained with a data set of bug fixes to then reintroduce the fixed bugs in similar code structures. This could lead to more realistic mutations.

Papers: [3, 4]

Tools: —

**Present a limited amount of mutants:** In a code review, for example, a developer might be overwhelmed if a large set of mutants is presented to them simultaneously. This could be mitigated by only presenting a subset of mutants, either a random set or the most likely to be productive based on some metric.

Papers: [38]

Tools: —

**Only mutate code covered by tests:** Only the lines of code covered by tests can be killed. Hence, mutating code not covered by tests is pointless.

Papers: [38]

Tools: Dextool, Mull

**Specify tests to use:** Limit mutation to lines of code covered by a specific test suite. Sometimes, we might be interested in increasing the test quality of a particular test suite. Mutating anything else in this scenario would be a waste of time.

Papers: [38]

Tools: Dextool, Mull

**Flag mutant as irrelevant:** Some mutants are unproductive and may not improve

the test suite when killed. They could be manually flagged as irrelevant to avoid having this mutant appear in the mutation testing result in future runs. Similarly, useful mutants could be manually flagged as helpful. This could be used to build historical data on the mutant operators that usually produce good mutants for certain good structures. ignored or avoided in future runs

Papers: [38]

Tools: Dextool

**Specify code to mutate:** Limit mutation generation to certain code lines. For example, a developer wants to improve the test quality of a certain area of the code.

Papers: —

Tools: Mutate++, CCmutator

**Stop after a number of live mutants:** The user might be interested or may not have time to view more than a certain number of live mutants. We could then stop after that specific number of live mutants have been detected.

Papers: —

Tools: Dextool

**Skip killed mutants:** If the lines of code were mutation tested in the last run and the tests covering that code have not changed, then it is a waste of time to mutate and test that code again. The mutation result could be stored between runs, so that killed mutants can be skipped if there has been no change to the test suite.

Papers: —

Tools: Dextool

**Detect redundant tests:** Test that do not uniquely kill any mutant indicates that it has a lot of overlap with other tests. These tests could be flagged so they can be investigated if they are redundant or not.

Papers: —

Tools: Dextool

**Resume mutation testing:** Due to the nature of mutation testing, it can take a long time to complete. The mutants generated and mutation result could be stored continuously so that if, for whatever reason the mutation testing is stopped, it can be resumed. For example, pause the mutation testing to yield the computational power to another process.

Papers: —

Tools: Dextool, Mutate++

**Timeout tests:** Mutations might lead to infinite loops. One way to detect these loops is timeouts, and the timeout is usually set as a standard value or dynamically set for every test suite based on their normal run time.

Papers: —

Tools: Dextool, Mull, Mutate++

**Parallel mutation testing:** An effective way to speed up mutation testing without

impacting the testing quality is parallelization. For example, mutants can be tested in parallel on different sub-process.

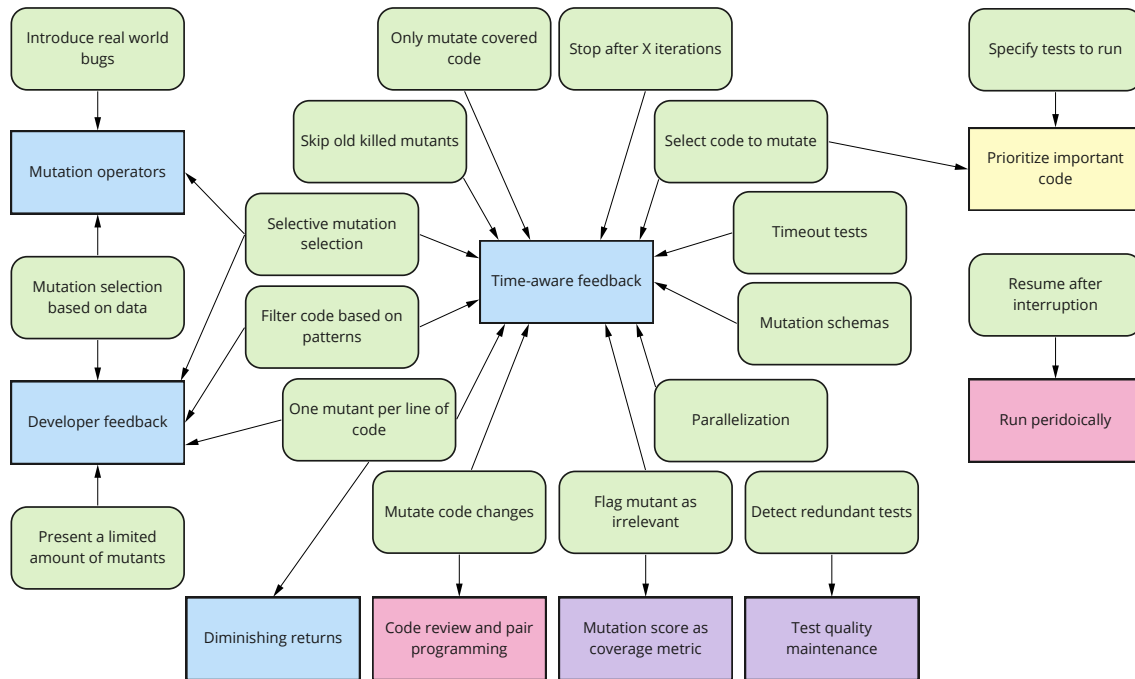
Papers: [38]

Tools: Dextool, Mull

**RQ2.2 (Techniques for effective mutation testing):** A common theme between the identified techniques is that they try to reduce the time spent on mutation testing while still maintaining most of the quality. Examples of this are; trying to reduce the number of mutants while still keeping the most effective ones, reducing time spent testing unproductive mutants, and parallelization of mutation tools.

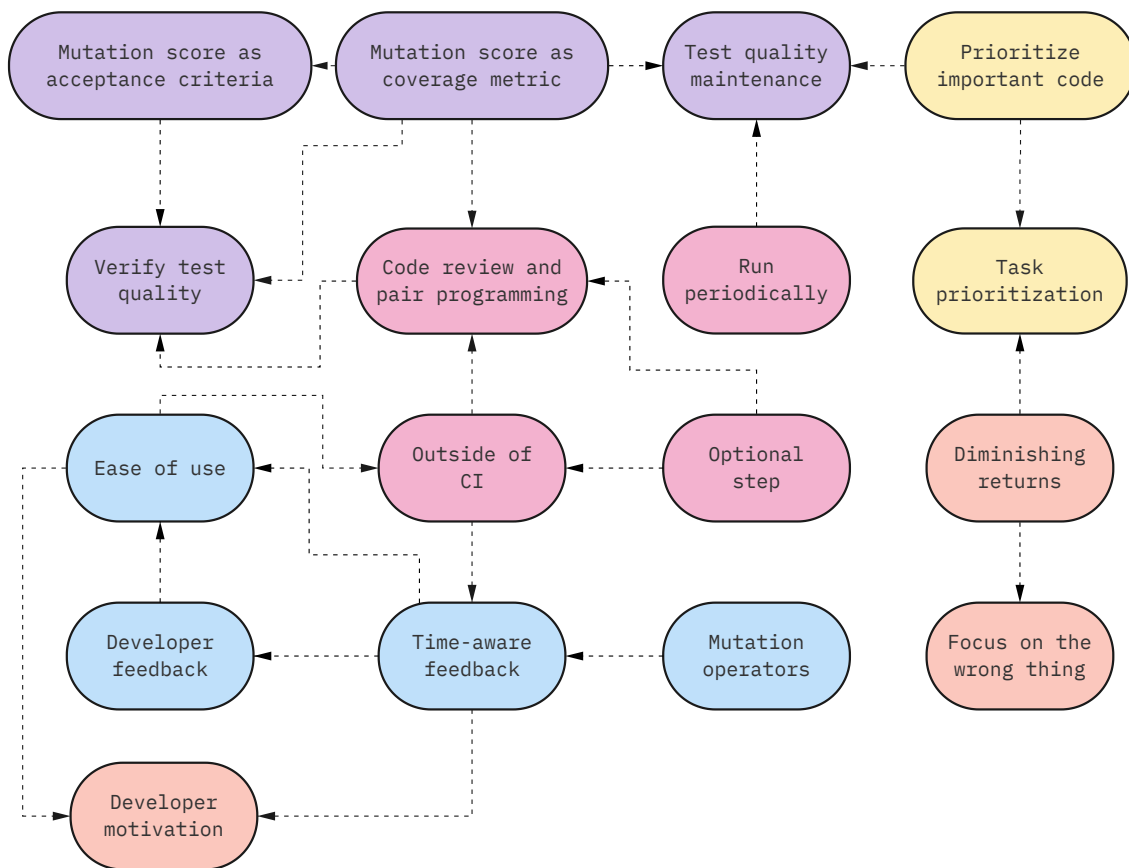
### 5.2.3 Guidelines for Mutation Testing in CI (RQ2.3)

In this section, we will analyze the results of connecting **RQ2.1** and **RQ2.2** to devise guidelines to apply mutation testing in a CI context. The resulting connections from the matching of the techniques and practice of **RQ2.1** to the sub-themes of **RQ2.2** can be seen in Figure 5.4 and the relation between the sub-themes themselves in Figure 5.5.



**Figure 5.4:** Technique/practice and sub-theme match map. Rectangles are sub-themes, and rectangles with rounded edges are techniques/practices from the literature or tools. Themes are color-coded; purple = quality, blue = implementation, pink = when to use, and yellow = prioritization.

**Time-aware feedback:** Developers want feedback as soon as possible. As shown in Figure 5.4 most of the techniques in the literature are aimed at addressing this topic. Techniques that speed up mutation testing without affecting mutants should be



**Figure 5.5:** Sub-theme relational map, this map is separate from the theme overview to reduce clutter. Themes are color-coded; purple = quality, blue = implementation, pink = when to use, yellow = prioritization, and orange = risks.

applied, such as *parallelization*. Furthermore, mutation testing should be accelerated by using techniques to reduce the number of mutants while still keeping the number of productive mutants high. This could be done by applying techniques such as only generating *one mutant per line of code*, since generating more generally has *diminishing returns*.

**Test quality maintenance:** Mutation testing can be used to maintain the quality of the tests, as the interviewees mentioned. This can be done by using *mutation scores as a coverage metric* to identify areas of the code that might lack test quality. Furthermore, when applying mutation testing to a large codebase, the essential areas of the code could be *prioritized* for mutation testing. Additionally, mutation testing could run *run periodically* when there is free machine time on CI/CD machines to not block more critical jobs.

**Mutation testing at code check-in:** Mutation testing could be used as a *optional* or *mandatory* quality check for new pull requests in a CI/CD pipeline. This could be as part of a *code review* process or as a sanity check after a code check-in, for example, after a *pair programming* session. Only the *new code should be mutated*, as at code check-in, since we are looking for feedback on the new code. For instance, the

resulting mutation score or the number of live mutants could be used *as a coverage metric* to judge the quality of the tests for the new code.

**Developer motivation:** The mutation tool should be *trivial to use* as a part of the CI/CD pipeline as any interruption in the workflow caused by a tool could lead the developer to be less motivated to use the tool. Furthermore, the tool should *work correctly* and provide *minimal false positives*, as a failing tooling or false positives can lead to frustration. For example, code could be filtered based on patterns to not mutate lines of code that lead to false positives most of the time.

Additionally, the *mutation reduction techniques* mentioned for time-aware feedback can, if configured correctly, in addition to producing faster mutation results, also have better mutants on average than when not filtering mutants since more of the *less effective ones have been filtered out*. This means that the developer can get *faster feedback* and a *higher percentage of effective mutants*.

**Risks:** There are some risks when using mutation testing. Inexperience can lead to wrong developer focus, wasting precious development time. For example, the developer may be led to focus on structural-based metrics and neglect to check whether the actual functionality is being tested. To avoid problems like these, developers who are inexperienced with mutation testing should be appropriately trained to understand the mutation result better. Similarly, the initial setup of the mutation testing tool might require a lot of effort due to inexperience and potentially lacking tools.

**RQ2.3 (Guidelines for mutation testing in CI):** We have identified recommendations by cross-referencing techniques from literature and factors mentioned by practitioners. Our recommendations are, among others; The mutation testing should interfere as little as possible with the developer, essential areas of the code should be prioritized for mutation testing, developers inexperienced with mutation testing should be trained, less time-critical mutation testing should wait until there is free machine time to run, and there should be an option to run mutation testing on a local machine.





# 6

## Discussion

### 6.1 Mutation tool summary

The tools compared in this thesis have different strengths and weaknesses. This section will give a short summary of the main advantages and disadvantages of each tool.

#### 6.1.1 Dextool

Dextool was the most mature mutation tool that we examined. It had the fewest bugs, and up-to-date documentation, and is being actively maintained. It was also the only mutation tool that could perform mutation testing based on a git diff, potentially greatly speeding up mutation testing for new code commits. Although the tool documentation was up to date, it was sometimes hard to find information since some of the documentation was in the generated configuration files and not in the actual documentation files. The tool also has partial support for mutation schema; however, as mentioned in the documentation for the tool, the probability of the tools' mutation schema failing increases the more mutants are in them, leading to the mutants often having to be tested individually, causing a long test execution phase.

#### 6.1.2 Mull

Mull was the fastest mutation testing tool due to its ability to inject all mutants at once and then activate them one at a time without the need to recompile when swapping the mutant under test, as traditional mutation testing tools do. But the fact that the tool only accepts one test binary at a time can cause some initial effort to get the overall project mutation score. Parts of the project test suit could have to be rewritten so that the tests can be compiled into a single test binary. The tool also had some bugs and out-of-date documentation that did not reflect the latest tool version; for example, a function had been removed from the tool but was still in the documentation, and a new configuration option was not yet added to the documentation.

### 6.1.3 MuCPP

MuCPP offered the most varied mutation operators, with support for both traditional mutation operators and language-specific operators. The tool does however require a Makefile with specific rules to function and the creation of a test library that reports the test results in a specific way to perform the mutation test execution, making the tool require a lot of initial effort to set up the tool for a project. The tool is also the only tool we looked at that is not open-source and has no changelog on its website, making it hard to judge if the tool is still being actively maintained. Furthermore, the licensing of the software is not clear.

### 6.1.4 Mutate++

Mutate++ was the only tool that offered the ability to select which lines of code to mutate. The tool does not offer any CLI interface or other way to use it programmatically, making it inadequate for CI use. We were unable to run mutation tests using the tool due to an error being raised, as explained in 5.1.1. But Mutate++ is still in a very early stage of its development, as stated by its owner [31] and is actively maintained, so it could be worth revisiting in the future.

### 6.1.5 CCmutator

CCMutator was the only tool that offered multi-threaded mutation operators, which could potentially be of interest for multi-threaded applications. However, we could not generate mutants with the tool due to an error explained in 5.1.1, and the tool has been seemingly abandoned; it was last updated on 27 September 2013.

**RQ1 (Evaluation of C++ Mutation tools):** Dextool and Mull were considered the readiest for use in practice and are actively maintained. MuCPP could be used in practice if the language-specific operators it offers are desired, but the licensing should be investigated. Mutate++ is very early in its development, and CCmutator is seemingly abandoned; both were deemed unsuitable for use in practice.

## 6.2 Identifying tools suitable for CI integration

The evaluated mutation testing tools varied in their ability to integrate into a continuous integration workflow. We decided to rule out two tools as not suitable for CI integration early in the process. As mentioned in Chapter 5, we were unable to generate mutants using CCmutator, and the tool was last updated on 27 September 2013. These two factors led us to not consider CCmutator suitable for integration with CI. Furthermore, Mutate++ reliance on a GUI —lacking CLI support— led us to not consider the tool for integration with CI.

This left us with three tools to consider, which are Dextool, Mull, and MuCPP. We identified three reasons to not recommend MuCPP. First, MuCPP requires a Makefile with specific rules and the creation of a test library that reports the test

results in a specific way back to the tool when performing the mutant test execution. Second, the tool has no changelog on the tool’s website or version number of the tool, making it hard to grasp how often the tool has been updated. Finally, on the website, it is unclear what copyright the project has. Therefore, it’s not clear if the tool is allowed to be used in a commercial setting.

The two remaining tools, Dextool and Mull, can still be considered for CI integration, as both have their own benefits and drawbacks. The mutation operators of each tool might have to be taken into consideration when deciding on a tool to use since it might be that specific operators are favorable in certain circumstances. However, if mutation operators are disregarded, then under certain circumstances there are clear advantages for specific tools. On the one hand, if we want a tool to generate mutants and have another tool performing the mutation testing, mutate the git diff for incremental changes, or if we want to stop testing after a specific number of live mutants, then Dextool is our recommended tool. On the other hand, if it is possible to compile all tests into the same binary, or alternatively if we want the fastest feedback for a specific test suite and we are not interested in the overall mutation score, then Mull is the recommended tool.

**RQ1 (Evaluation of C++ Mutation tools):** Only Dextool and Mull were considered suitable for CI integration. The other tools are not suitable for multiple reasons, including lack of CLI support, various bugs, lack of ongoing support, or lack of clarity in licensing.

### 6.3 Possible tool improvements

The mutation tools evaluated varied greatly in the features and operators they supported, as seen in Chapter 5. Figure 5.9 could be used to identify valuable features to develop for future iterations of the mutation tools. For example, adding the feature to limit mutations to one mutant per line of code to Dextool and Mull, or the ability for Mull to pause mutation testing and/or pick up from where it stopped after an interruption.

### 6.4 Compiler Compatibility

Four out of the five mutation tools depended on Clang and, therefore, were closely related to the compiler version. We were unable to integrate any of the mutation tools examined with the toolchain at the partner company due to incompatible compiler versions, as mentioned in Section 6.2. This was an important revelation since one could easily forget to consider the maintenance burden of introducing a new tool in CI. The compiler version would have to be in lockstep with the tool version to support any of these four mutation tools in CI. Hence, an uplift in the compiler version would require an uplift of the mutation tool version if such a version even exists. For example, none of the four tools supported Clang 14, which is the latest version of Clang as of writing.

While a separate toolchain could be maintained to support the older compiler version required to run the tool, this would add technical debt as the code now would have to be compatible with multiple compiler versions. For example, the partner company’s toolchain relied on Clang 14 specific features to compile the code, potentially meaning that the code would have to be re-worked to work an older Clang version.

## 6.5 Threats to Validity

The threats to validity can be distinguished between four aspects of validity [41]—construct validity, internal validity, external validity, and reliability. In this section, the four aspects will be briefly described, along with threats to this thesis and mitigations that fall into these aspects. Furthermore, conclusion validity will also be discussed.

### 6.5.1 Construct Validity

Construct validity is to what extent the means of interpreting a phenomenon represent what the researcher has in mind [42]. For example, there could be different understandings of what a term implies between the researcher and the interviewee. In this study, that could be different understandings of what the term mutation testing suggests, if one is not already familiar with the concept. To mitigate this problem, interviewees were offered an introduction to the topic before starting the interview.

### 6.5.2 Internal Validity

Internal validity is the threat to data collection caused by alternative explanations. An example of this is that the reliability of the benchmark data for the mutation tools is impacted by the fact that only one test binary could be provided at a time when performing mutation testing. This was due to a limitation of the Mull tool. This led to the use of only a subset of the test suite for projects that did not compile all of their tests into one test binary. This shortened the test execution time but kept the compile-time essentially the same. However, the result is still valid since the same tests were supplied to all the tools; the only difference is that the project is treated like it has lower test coverage.

Another threat to internal validity was that the available literature and documentation on the inner workings of the evaluated mutation testing tools differed significantly; only Mull and MuCPP have papers that describe how they work. While the engineering details of individual tools are not the focus of this thesis, the problem with a lack of documentation is that it hinders understanding. It makes it hard to explain why its performance was what it was, or to understand the factors that affect its performance. Due to this, it makes it harder to compare tools, if each tool is not well documented. However, the results are still valid since the tool’s overall performance is what is being measured.

### 6.5.3 External Validity

External validity is to what extent the result can be generalized to be of interest to people not in the study. An external validity of the thesis is that it was limited to the context of the partner company Zenseact, which limits the generalizability of the result. However, we believe that our findings will be applicable to other companies operating in similar domains.

Moreover, we could not integrate any of the mutation tools with the toolchain in the partner company due to incompatible Clang versions. This meant that we could not test the applicability of the evaluated mutation testing tools in practice at the partner company. However, two CI workflows were created for the TinyXML-2 project as proof-of-concept for using mutation tools in CI. The workflows are specific to TinyXML-2 but can easily be adapted to other projects.

### 6.5.4 Reliability

Reliability is the degree to which the result and the researchers depend on each other. A threat to reliability in this thesis is the thematic analysis of the interviews. To mitigate this risk, two supervisors were involved in triangulating the result of the interview coding process.

Similarly, the interview instrument in the form of an interview guide created by the researcher represents a threat to reliability. This risk was mitigated in three ways. Firstly, the interviews were semi-structured to allow interviewees to present new ideas. Secondly, a supervisor reviewed the instrument before use. Thirdly, the interview guide was iterated upon as needed between the interviews.

### 6.5.5 Conclusion Validity

Conclusion validity is to the extent that the conclusion is correct based on the data. The limited number of interviews presents a threat to the correctness of the conclusion of this thesis. The number of candidates for the interviews was sparse due to limited experience with mutation testing at the partner company, since it's not a widely adopted practice within the industry. This led to no theoretical saturation, the last interview still added more findings. However, as the available interview candidates found by company wide communications was saturated, the resulting data represents the context of the partner company of which the study is limited to.

## 6.6 Future Work

Some of the mutation tools examined in this thesis are actively maintained and improved. These tools could be revisited in the future to see how they have improved and how they could still be improved. Alternatively, new functionality could be developed for a tool to allow it to utilize more of the identified techniques for mutation testing from the literature. The improvement could then be measured to see the

impact of the techniques. Similarly, techniques from the existing mutation tools not represented in the literature could be further explored and tested.

It could be interesting to compare the features offered by C++ mutation tools with those of other languages. To see how they differ and, in doing so, identify what they can learn from each other.

Future research could investigate developers' goals for effective mutation testing from other contexts to see if their views differ.

To solve the problem of mutation tools falling out of sync with the latest compiler version, the possibility of integrating mutation testing as part of the LLVM project could be explored. Thus, always keeping the mutation test tool in sync with the latest compiler version. Similarly to how the clang-based C++ linter tool clang-tidy is a part of the LLVM project [6].

The proof-of-concept GitHub Actions workflow could be extended to work with a wider variety of projects, or even make it so other mutation tools could be slotted in as alternatives to the two supported tools.

# 7

## Conclusion

This thesis consisted of research to identify guidelines for using mutation testing as part of a continuous integration pipeline.

The thesis was carried out by performing an empirical evaluation of mutation testing tools for C++, conducting semi-structured interviews with stakeholders at a partner company, and conducting a literature review. The results were then combined to identify the guideline for mutation testing within CI.

The capabilities of the evaluated C++ mutation testing tools are listed below:

- Dextool offers the most features and is the only tool capable of mutating a git diff. Nevertheless, the tool was not the fastest to perform mutation testing.
- Mull was by far the fastest at performing the mutation testing. But the tool can only use one test binary at a time.
- MuCPP offers the most varied mutation operators and is the only tool that offers class-level mutation operators. However, it is the only tool evaluated that is not open-source.
- Mutate++ is very early in its development and cannot be used programmatically.
- CCmutator is the only tool to offer multi-threaded mutation operators. However, the tool seems to have been abandoned.

We can only recommend using Dextool or Mull for CI integration. A proof-of-concept mutation testing workflow was created to demonstrate the use of the two tools in a CI environment.

How mutation testing can be best used within continuous integration:

- The mutation testing tool should interfere as little as possible with the developer.
- There should be an option to run the mutation testing on a local machine

outside the CI/CD pipeline.

- The subset of mutants to use should be optimized to provide timely feedback while maintaining effectiveness.
- Less time-critical feedback such as mutation results for old code can wait until there is free machine time.
- When applying mutation testing on a large existing codebase, essential areas of the code should be prioritized.
- Developers inexperienced with mutation testing should be trained to understand the mutation result better.

The results of this study will provide guidance to practitioners who want to make mutation testing a part of their CI workflow. Furthermore, this study will show where future work can improve existing C++ mutation testing tools to perform mutation testing effectively. Future research on this topic could include evaluating the capabilities of mutation tools for other programming languages to see what they offer. Furthermore, future studies could investigate whether the developer's goals for effective mutation testing are similar in different contexts, or the proof-of-concept GitHub Actions workflow could be extended to work with a wider variety of projects and mutation tools.



# Bibliography

- [1] Richard Baker and Ibrahim Habli. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *IEEE Transactions on Software Engineering*, 39(6):787–805, 2013.
- [2] Orit Baruch and Shmuel Katz. Partially interpreted schemas for csp programming. *Science of Computer Programming*, 10(1):1–18, 1988.
- [3] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. What it would take to use mutation testing in industry—a study at facebook. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021.
- [4] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. The care and feeding of wild-caught mutants. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.
- [5] Timothy Budd. *Mutation analysis of Program Test Data*. PhD thesis, Yale University, 1980.
- [6] Clang. Clang-tidy. <https://clang.llvm.org/extra/clang-tidy/>.
- [7] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Francisco Palomo-Lozano, Antonio García-Domínguez, and Juan José Domínguez-Jiménez. Assessment of class mutation operators for C++ with the mucpp mutation system. *Information and Software Technology*, 81:169–184, 2017.
- [8] Pedro Delgado-Pérez, Sergio Segura, and Inmaculada Medina-Bulo. Assessment of c++ object-oriented mutation operators: A selective mutation approach. *Software Testing, Verification and Reliability*, 27(4-5), 2017.
- [9] R.a. Demillo, R.j. Lipton, and F.g. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.
- [10] A. Denisov and S. Pankevich. Mull it over: Mutation testing based on llvm. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 25–31, April 2018.

- [11] Alex Denisov and Stanislav Pankevich. Mull it over: Mutation testing based on llvm. *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2018.
- [12] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015.
- [13] fmtlib. Fmt. <https://github.com/fmtlib/fmt>.
- [14] Gregory Gay, Matt Staats, Michael Whalen, and Mats P. Heimdahl. The risks of coverage-directed test case generation. *IEEE Transactions on Software Engineering*, 41(8):803–819, 2015.
- [15] GitHub, Inc. Github actions. <https://docs.github.com/en/actions>.
- [16] IEEE Xplore. Ieee xplore. <https://ieeexplore.ieee.org/Xplore/home.jsp>.
- [17] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. *Proceedings of the 36th International Conference on Software Engineering*, 2014.
- [18] jbeder. yaml-cpp. <https://github.com/jbeder/yaml-cpp>.
- [19] joakim-brannstrom. Dextool mutate. <https://github.com/joakim-brannstrom/dextool/tree/master/plugin/mutate>.
- [20] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014*, page 437–440, New York, NY, USA, 2014. Association for Computing Machinery.
- [21] Markus Kusano and Chao Wang. Ccmulator: A mutation generator for concurrency constructs in multithreaded c/c applications. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 722–725, 2013.
- [22] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis& transformation. *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2014.
- [23] leethomason. Tinyxml-2. <https://github.com/leethomason/tinyxml2>.
- [24] Richard Lipton. Fault diagnosis of computer programs. *none*, 1971.
- [25] LLVM Foundation. Clang: a c language family frontend for llvm. <https://clang.llvm.org/>.

- [26] Wei Ma, Thierry Titcheu Chekam, Mike Papadakis, and Mark Harman. Mudelta: Delta-oriented mutation testing at commit time. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021.
- [27] Pedro Reales Mateo and Macario Polo Usaola. Mutant execution cost reduction: Through music (mutant schema improved with extra code). *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012.
- [28] mosra. Corrade. <https://github.com/mosra/corrade>.
- [29] mull project. Mull 0.16.0 documentation. <https://mull.readthedocs.io/en/0.16.0/>.
- [30] nlohmann. Json. <https://github.com/nlohmann/json>.
- [31] nlohmann. Mutate++. [https://github.com/nlohmann/mutate\\_cpp](https://github.com/nlohmann/mutate_cpp).
- [32] A.j. Offutt and S.d. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, 1994.
- [33] orgardj. Mutation tool settings. [https://github.com/Orgardj/mutation\\_testing/](https://github.com/Orgardj/mutation_testing/).
- [34] orgardj. Pof tiny-xml2 ci workflow. [https://github.com/Orgardj/tinyxml2/tree/mutation\\_testing/](https://github.com/Orgardj/tinyxml2/tree/mutation_testing/).
- [35] Mike Papadakis, Marcio Delamaro, and Yves Le Traon. Mitigating the effects of equivalent mutants with mutant classification strategies. *Science of Computer Programming*, 95:298–319, 2014.
- [36] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: An analysis and survey. *Advances in Computers*, 112:275–378, 2019.
- [37] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and Rene Just. Does mutation testing improve testing practices? *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021.
- [38] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and Rene Just. Practical mutation testing at scale: A view from google. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.
- [39] Goran Petrovic, Marko Ivankovic, Bob Kurtz, Paul Ammann, and Rene Just. An industrial application of mutation testing: Lessons, challenges, and research directions. *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2018.
- [40] Rudolf Ramler, Thomas Wetzlmaier, and Claus Klammer. An empirical study

- on the application of mutation testing for a safety-critical industrial software system. *Proceedings of the Symposium on Applied Computing*, 2017.
- [41] Colin Robson. *Real world research: a resource for social scientists and practitioner-researchers*. Blackwell, 2002.
- [42] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2008.
- [43] Pankaj Saha, Angel Beltre, Piotr Uminski, and Madhusudhan Govindaraju. Evaluation of docker containers for scientific workloads in the cloud. *Proceedings of the Practice and Experience on Advanced Research Computing*, 2018.
- [44] timsort. Timsort. <https://github.com/timsort/cpp-TimSort>.
- [45] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. Mutation analysis using mutant schemata. *Proceedings of the 1993 international symposium on Software testing and analysis - ISSTA '93*, 1993.
- [46] Volvo Cars. Volvo cars appoints Ödgård andersson ceo of its autonomous drive software company zenseact. <https://www.media.volvocars.com/global/en-gb/media/pressreleases/272347/volvo-cars-appoints-odgard-andersson-ceo-of-its-autonomous-drive-software-comp>
- [47] Miguel Ángel Álvarez García. Automation and evaluation of mutation testing for the new c++ standards. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 150–152, 2021.

# A

## Mutation Operators

### A.1 MuCPP Class-level Mutation Operators

**Table A.1:** Class-level mutation operators supported by MuCPP.

Operator	Name
IHI	Hiding variable insertion
IHD	Hiding variable deletion
ISD	Base keyword deletion
ISI	Base keyword insertion
IOD	Overriding method deletion
IOP	Overriding method calling position change
IOR	Overridden method rename
IPC	Explicit call of a parents constructor deletion
IMR	Multiple inheritance replacement
PVI	"virtual" modifier insertion
PCD	Type cast operator deletion
PCI	Type cast operator insertion
PCC	Cast type change
PPD	Parameter variable declaration with child class type
PMD	Member variable declaration with parent class type
PNC	"new" method call with child class type
OMD	Overloading method deletion
OMR	Overloading method contents replace
OAN	Argument number change
OAQ	Argument order change
MCO	Member call from another object
MCI	Member call from another inherited class
EHC	Exception handling change
EHR	Exception handler removal
CTD	"this" keyword deletion
CTI	"this" keyword insertion
CID	Member variable initialisation deletion

Operator	Name
CDC	Default constructor creation
CDD	Destructor method deletion
CCA	Copy constructor and assignment operator overloading deletion

## A.2 CCmutator Multi-threaded Mutation Operators

**Table A.2:** Multi-threaded mutation operators supported by CCmutator

Operator	Name
MSEM	Modify Permit Count in Semaphore
MWAIT	Modify parameter in cond_timedwait()
MCNT	Modify cond_timedwait() time value
RMWAIT	Remove Call to cond_wait()
RMWAIT	Remove Call cond_timedwait()
SWPTW	Swap cond_timedwait() with cond_wait()
RMSIG	Remove Call cond_signal()
RMSIG	Remove Call to cond_broadcast()
SWPB	Swap cond_signal() with cond_broadcast()
SWPS	Swap cond_broadcast() with cond_signal()
RMJOIN	Remove Call to join()
RMYIELD	Remove call to yield()
REPJN	Replace join() with sleep()
RMVOL	Remove Volatile Keyword
SWPLCK	Swap lock-unlock pairs
RMECS	Remove explicit critical section
SHFECS	Shift explicit Critical Region
SHKECS	Shrink explicit Critical Region
EPDECS	Expand explicit Critical Region
SPLTECS	Split Critical Region
RMF	Remove memory fence
MFE	Modify memory fence ordering constraint
REPSF	Replace single-thread sync fence with cross-thread
REPCF	Replace cross-thread sync fence with single-thread
REPAL	Replace atomic load with non-atomic load
MAL	Modify atomic load ordering constraint
REPSL	Replace single-thread sync atomic load with cross-thread
REPCL	Replace cross-thread sync atomic load with single-thread
RELAS	Replace atomic store with non-atomic store
MAS	Modify atomic store ordering constraint
REPSS	Replace single-thread sync atomic store with cross-thread

Operator	Name
REPCS	Replace cross-thread sync atomic store with single-thread
MRMW	Modify atomic read-modify-write ordering constraint
RSRM	Replace single-thread sync atomic read-modify-write with cross-thread
RCRM	Replace cross-thread sync atomic read-modify-write with single-thread
MCX	Modify compare exchange ordering constraint
RSCX	Replace single-thread sync compare exchange with cross-thread
RCCX	Replace cross-thread sync compare exchange with single-thread

### A.3 Mutation Operator Examples

**Table A.3:** Mutation operators with examples

Operator	Name	Example
AOR (ARS, ASR)	Assignment operator replacement	$a += b \rightarrow a -= b$
AOR (ARB, ARU)	Arithmetic operator replacement	$a + b \rightarrow a - b$
DCR	Decision/ condition requirement	$\text{if } (x) \rightarrow \text{"if(true)" "if(false)"}$
ROR	Relational operator replacement	$a < b \rightarrow a > b$
SDL	Statement deletion	$c = a + b \rightarrow$
ABS	Absolute value insertion	$c = a + b \rightarrow c = \text{abs}(a + b)$
COR	Conditional operator replacement	$a    b \rightarrow a \&\& b$
LCR (LOR)	Logical connector (operator) replacement	$a   b \rightarrow a \& b$
UOI	Unary operator insertion	$x \rightarrow !x$
CR	Constant replacement	$\text{const int } a = 42 \rightarrow \text{const int } a = 0$
SVR	Scalar variable replacement	$a = 0 \rightarrow a = 42$
SBR	Statement block removal	<pre> <b>if</b> (a)     b = a;  → </pre>
AOI (AIU, AIS)	Arithmetic operator insertion	$a = b \rightarrow a = -b$
ADS	Arithmetic operator deletion	$b = a++ \rightarrow b = a$
COD	Conditional operator deletion	$a != b \rightarrow a = b$
COI	Conditional operator insertion	$a = b \rightarrow a != b$

## A. Mutation Operators

---

Operator	Name	Example
IHI	Hiding variable insertion	<pre> class A {     ...     int a; } class B : public A {     ... }  →  class A {     ...     int a; } class B: public A {     ...     int a; } </pre>
IHD	Hiding variable deletion	<pre> class A {     ...     int a; } class B: public A {     ...     int a; }  →  class A {     ...     int a; } class B: public A {     ... } </pre>
ISD	Base keyword deletion	return A::a; → return a;
ISI	Base keyword insertion	return a; → return A::a;
IOD	Overriding method deletion	<pre> class A {     ...     void m (int a) { ... }; } class B: public A {     ...     void m (int a) { ... }; }  →  class A {     ...     void m (int a) { ... }; } class B: public A {     ... } </pre>



Operator	Name	Example
IOP	Overriding method calling position change	<pre>class A {   ...   void m() {     a = 5;     ...   } } class B: public A {   ...   void m() {     super.m();     a = 10;     ...   } } → class A {   ...   void m() {     a = 5;     ...   } } class A: public B {   ...   void m() {     a = 10;     super.m();     ...   } }</pre>
IOR	Overridden method rename	<pre>class A {   ...   virtual void m1() { <i>//body of m1</i> }   void m2() { ... m2(); } } → class A {   ...   virtual void m1() { <i>//body of m1</i> }   virtual void m3() { <i>//body of m1</i> }   void m2() { ... m3(); } }</pre>
IPC	Explicit call of a parents constructor deletion	<pre>class B : public A {   ...   B (int a) {     A(a);     ...   } } → class B : public A {   ...   B (int a) {     ...   } }</pre>

---

## A. Mutation Operators

Operator	Name	Example
IMR	Multiple inheritance replacement	<pre> <b>class</b> C: <b>public</b> A, <b>public</b> B {     ...     <b>void</b> m() {         b = A::a + 1;     } } → <b>class</b> C: <b>public</b> A, <b>public</b> B {     ...     <b>void</b> m() {         b = B::a + 1;     } } </pre>
PVI	"virtual" modifier insertion	<code>void m() {...} → virtual void m() {...}</code>
PCD	Type cast operator deletion	<code>a = (int)b → a = b</code>
PCI	Type cast operator insertion	<code>a = b → a = (int)b</code>
PCC	Cast type change	<code>a = (int)b → a = (double)b</code>
PPD	Parameter variable declaration with child class type	boolean equals (Child o) → boolean equals (Parent o)
PMD	Member variable declaration with parent class type	Child a → Parent a
PNC	"new" method call with child class type	<code>a = new Parent() → a = new Child()</code>
OMD	Overloading method deletion	<pre> <b>void</b> x (<b>int</b> i) { ... } <b>void</b> x (<b>float</b> i) { ... } → <b>void</b> x (<b>float</b> i) { ... } </pre>
OMR	Overloading method contents replace	<pre> <b>void</b> x (<b>int</b> i) { ... } <b>void</b> x (<b>int</b> i, <b>int</b> j) { ... } → <b>void</b> x (<b>int</b> i) { ... } <b>void</b> x (<b>int</b> i, <b>int</b> j) { x(i) } </pre>
OAN	Argument number change	<code>void x(a, b) → void x(a)</code>
OAQ	Argument order change	<code>void x(a, b) → void x(b, a)</code>
MCO	Member call from another object	<pre> <b>class</b> B {     A a;     A b;     <b>void</b> m2() {... a.m1() } } → <b>class</b> B {     A a;     A b;     <b>void</b> m2() {... b.m1() } } </pre>

Operator	Name	Example
MCI	Member call from another inherited class	<pre> <b>class</b> C: <b>public</b> A {     ... } <b>class</b> B: <b>public</b> A {     A a;     C b;     <b>void</b> m2() { ... a.m1() } } → <b>class</b> C: <b>public</b> A {     ... } <b>class</b> B: <b>public</b> A {     A a;     C b;     <b>void</b> m2() { ... b.m1() } } </pre>
EHC	Exception handling change	<code>catch(e) { ... } → catch(e) { throw }</code>
EHR	Exception handler removal	<code>catch(e) { ... } →</code>
CTD	"this" keyword deletion	<code>this- &gt; x = x → x = x</code>
CTI	"this" keyword insertion	<code>x = x → this- &gt; x = x</code>
CID	Member variable initialisation deletion	<code>A :: A() : a(0){b = 1; } → A :: A() : a(0){}</code>
CDC	Default constructor creation	<pre> <b>class</b> A {     A(<b>int</b> a) { ... } } → <b>class</b> A{     A()     A(<b>int</b> a) { ... } } </pre>
CDD	Destructor method deletion	<code>A(){...} →</code>
CCA	Copy constructor and assignment operator overloading deletion	<pre> A(<b>const</b> A&amp; copy) { ... } A&amp; <b>operator</b> = (<b>const</b> A&amp; copy) { ... } → A&amp; <b>operator</b> = (<b>const</b> A&amp; copy) { ... } </pre>
MSEM	Modify permit count in semaphore	<code>sem_init(sem, 1, 0) → sem_init(sem, 1, 1)</code>
MWAIT	Modify parameter in <code>cond_timedwait()</code>	
MCNT	Modify <code>cond_timedwait()</code> time value	<code>pthread_cond_timedwait(&amp;t.cond, &amp;t.mn, 5) → pthread_cond_timedwait(&amp;t.cond, &amp;t.mn, 10)</code>
RMWAIT	Remove call to <code>cond_wait()</code>	<code>pthread_cond_wait(&amp;t.cond, &amp;t.mn) →</code>
RMWAIT	Remove call <code>cond_timedwait()</code>	<code>pthread_cond_timedwait(&amp;t.cond, &amp;t.mn, 5) →</code>
SWPTW	Swap <code>cond_timedwait()</code> with <code>cond_wait()</code>	<code>pthread_cond_timedwait(&amp;t.cond, &amp;t.mn, 5) → pthread_cond_wait(&amp;t.cond, &amp;t.mn)</code>
RMSIG	Remove call <code>cond_signal()</code>	<code>pthread_cond_wait(&amp;cond) →</code>
RMSIG	Remove call to <code>cond_broadcast()</code>	<code>pthread_cond_broadcast(&amp;cond) →</code>
SWPB	Swap <code>cond_signal()</code> with <code>cond_broadcast()</code>	<code>pthread_cond_wait(&amp;cond) → pthread_cond_broadcast(&amp;cond)</code>
SWPS	Swap <code>cond_broadcast()</code> with <code>cond_signal()</code>	<code>pthread_cond_broadcast(&amp;cond) → pthread_cond_wait(&amp;cond)</code>

## A. Mutation Operators

---

Operator	Name	Example
RMJOIN	Remove call to join()	<code>pthread_join(th, NULL) →</code>
RMYIELD	Remove call to yield()	<code>pthread_yield() →</code>
REPJN	Replace join() with sleep()	<code>pthread_join(th, NULL) → sleep(1)</code>
RMVOL	Remove volatile keyword	<code>volatile int a → int a</code>
SWPLCK	Swap lock-unlock pairs	<pre>pthread_mutex_lock(&amp;m1) pthread_mutex_lock(&amp;m2) pthread_mutex_unlock(&amp;m2) pthread_mutex_unlock(&amp;m1) → pthread_mutex_lock(&amp;m2) pthread_mutex_lock(&amp;m1) pthread_mutex_unlock(&amp;m1) pthread_mutex_unlock(&amp;m2)</pre>
RMECS	Remove explicit critical section	<pre>pthread_mutex_lock(&amp;m) pthread_mutex_unlock(&amp;m) →</pre>
SHFECS	Shift explicit critical region	<pre>pthread_mutex_lock(&amp;m) foo () ... pthread_mutex_unlock(&amp;m) bar () → foo () pthread_mutex_lock(&amp;m) ... bar () pthread_mutex_unlock(&amp;m)</pre>
SHKECS	Shrink explicit critical region	<pre>pthread_mutex_lock(&amp;m) foo () ... pthread_mutex_unlock(&amp;m) bar () → foo () pthread_mutex_lock(&amp;m) ... pthread_mutex_unlock(&amp;m) bar ()</pre>
EPDECS	Expand explicit critical region	<pre>pthread_mutex_lock(&amp;m) foo () ... pthread_mutex_unlock(&amp;m) bar () → pthread_mutex_lock(&amp;m) foo () ... bar () pthread_mutex_unlock(&amp;m)</pre>

Operator	Name	Example
SPLTECS	Split critical region	<pre>pthread_mutex_lock(&amp;m) foo () ... bar () pthread_mutex_unlock(&amp;m)  →  pthread_mutex_lock(&amp;m) foo () pthread_mutex_unlock(&amp;m) ... pthread_mutex_lock(&amp;m) bar () pthread_mutex_unlock(&amp;m)</pre>
RMF	Remove memory fence	<code>std::atomic_thread_fence(std::memory_order_release) →</code>
MFE	Modify memory fence ordering constraint	<code>std::atomic_thread_fence(std::memory_order_acquire) → std::atomic_thread_fence(std::memory_order_release)</code>
REPSF	Replace single-thread sync fence with cross-thread	<code>llvm::FenceInst::setSynchScope(SingleThread) → llvm::FenceInst::setSynchScope(CrossThread)</code>
REPCF	Replace cross-thread sync fence with single-thread	<code>llvm::FenceInst::setSynchScope(CrossThread) → llvm::FenceInst::setSynchScope(SingleThread)</code>
REPAL	Replace atomic load with non-atomic load	<code>llvm::LoadInst::setOrdering(Acquire) → llvm::LoadInst::setOrdering(NotAtomic)</code>
MAL	Modify atomic load ordering constraint	<code>std::atomic_load_explicit(&amp;obj, std::memory_order_consume) → std::atomic_load_explicit(&amp;obj, std::memory_order_acquire)</code>
REPSL	Replace single-thread sync atomic load with cross-thread	<code>llvm::LoadInst::setSynchScope(SingleThread) → llvm::LoadInst::setSynchScope(CrossThread)</code>
REPCL	Replace cross-thread sync atomic load with single-thread	<code>llvm::LoadInst::setSynchScope(CrossThread) → llvm::LoadInst::setSynchScope(SingleThread)</code>
RELAS	Replace atomic store with non-atomic store	<code>llvm::StoreInst::setOrdering(Release) → llvm::StoreInst::setOrdering(NotAtomic)</code>
MAS	Modify atomic store ordering constraint	<code>std::atomic_store(1, std::memory_order_relaxed) → std::atomic_store(1, std::memory_order_release)</code>
REPSS	Replace single-thread sync atomic store with cross-thread	<code>llvm::StoreInst::setSynchScope(SingleThread) → llvm::StoreInst::setSynchScope(CrossThread)</code>
REPCS	Replace cross-thread sync atomic store with single-thread	<code>llvm::StoreInst::setSynchScope(CrossThread) → llvm::StoreInst::setSynchScope(SingleThread)</code>
MRMW	Modify atomic read-modify-write ordering constraint	<code>std::atomic&lt;long long&gt;::fetch_add(1, std::memory_order_relaxed) → std::atomic&lt;long long&gt;::fetch_add(1, std::memory_order_release)</code>
RSRM	Replace single-thread sync atomic read-modify-write with Cross-thread	<code>llvm::AtomicRMWInst::setSynchScope(SingleThread) → llvm::AtomicRMWInst::setSynchScope(CrossThread)</code>
RCRM	Replace cross-thread sync atomic read-modify-write with single-thread	<code>llvm::AtomicRMWInst::setSynchScope(CrossThread) → llvm::AtomicRMWInst::setSynchScope(SingleThread)</code>
MCX	Modify compare exchange ordering constraint	<code>std::compare_exchange_strong(&amp;obj, std::memory_order_acquire) → std::compare_exchange_strong(&amp;obj, std::memory_order_release)</code>
RSCX	Replace single-thread sync compare exchange with crossthread	<code>llvm::AtomicCmpXchgInst::setSynchScope(SingleThread) → llvm::AtomicCmpXchgInst::setSynchScope(CrossThread)</code>
RCCX	Replace cross-thread sync compare exchange with singlethread	<code>llvm::AtomicCmpXchgInst::setSynchScope(CrossThread) → llvm::AtomicCmpXchgInst::setSynchScope(SingleThread)</code>



# B

## Mull Mutation Operator Mapping

**Table B.1:** Mull operator mapping

Mull operator	Description	Mapping
cxx_add_assign_to_sub_assign	Replaces += with -=	Assignment operator replacement
cxx_and_assign_to_or_assign	Replaces &= with  =	Assignment operator replacement
cxx_div_assign_to_mul_assign	Replaces /= with *=	Assignment operator replacement
cxx_mul_assign_to_div_assign	Replaces *= with /=	Assignment operator replacement
cxx_or_assign_to_and_assign	Replaces  = with &=	Assignment operator replacement
cxx_rem_assign_to_div_assign	Replaces %= with /=	Assignment operator replacement
cxx_sub_assign_to_add_assign	Replaces -= with +=	Assignment operator replacement
cxx_xor_assign_to_or_assign	Replaces ^= with  =	Assignment operator replacement
cxx_assign_const	Replaces 'a = b' with 'a = 42'	Assignment operator replacement
cxx_init_const	Replaces 'T a = b' with 'T a = 42'	Assignment operator replacement
cxx_post_dec_to_post_inc	Replaces x with x++	Assignment operator replacement
cxx_post_inc_to_post_dec	Replaces x++ with x--	Assignment operator replacement
cxx_bitwise_not_to_noop	Replaces ~x with x	Assignment operator replacement
cxx_minus_to_noop	Replaces -x with x	Assignment operator replacement
cxx_lshift_assign_to_rshift_assign	Replaces «= with »=	Assignment operator replacement
cxx_lshift_to_rshift	Replaces «with »	Assignment operator replacement
cxx_rshift_assign_to_lshift_assign	Replaces »= with «=	Assignment operator replacement
cxx_rshift_to_lshift	Replaces »with «	Assignment operator replacement
cxx_add_to_sub	Replaces + with -	Arithmetic operator replacement
cxx_div_to_mul	Replaces / with *	Arithmetic operator replacement
cxx_mul_to_div	Replaces * with /	Arithmetic operator replacement
cxx_rem_to_div	Replaces % with /	Arithmetic operator replacement
cxx_sub_to_add	Replaces - with +	Arithmetic operator replacement
cxx_and_to_or	Replaces & with	Conditional operator replacement
cxx_or_to_and	Replaces   with &	Conditional operator replacement
cxx_remove_negation	Replaces !a with a	Conditional operator replacement
cxx_xor_to_or	Replaces ^with	Conditional operator replacement
negate_mutator	Negates conditionals !x to x and x to !x	Conditional operator replacement
cxx_logical_and_to_or	Replaces && with	Logical connector replacement

## B. Mull Mutation Operator Mapping

---

Mull operator	Description	Mapping
cxx_logical_or_to_and	Replaces    with &&	Logical connector replacement
cxx_eq_to_ne	Replaces == with !=	Relational operator replacement
cxx_ge_to_gt	Replaces >= with >	Relational operator replacement
cxx_ge_to_lt	Replaces >= with <	Relational operator replacement
cxx_gt_to_ge	Replaces > with >=	Relational operator replacement
cxx_gt_to_le	Replaces > with <=	Relational operator replacement
cxx_le_to_gt	Replaces <= with >	Relational operator replacement
cxx_le_to_lt	Replaces <= with <	Relational operator replacement
cxx_lt_to_ge	Replaces < with >=	Relational operator replacement
cxx_lt_to_le	Replaces < with <=	Relational operator replacement
cxx_ne_to_eq	Replaces != with ==	Relational operator replacement
scalar_value_mutator	Replaces zeros with 42, and non-zeros with 0	Scalar variable replacement
cxx_replace_scalar_call	Replaces call to a function with 42	Scalar variable replacement
cxx_remove_void_call	Removes calls to a function returning void	Statement deletion



# C

## Interview Guide

### Research Questions

- **RQ2:** How can mutation testing best be used within continuous integration?
  - **RQ2.1:** What do developers see as the most effective use of mutations in their practice?
  - **RQ2.2:** Can we identify techniques meet the goals of the developers?
  - **RQ2.3:** Can general practices for the use of mutation testing within CI be identified?

### Section 1 - Introduction

Interviewer introduces the goal of the study (integrate mutation testing into the workflow of developers and testers), and asks the participant for consent. Particularly, ask for consent to record the interview.

### Section 2 - Questions about integrating mutation testing into the workflow

*Set of Question 1: Description of what mutation testing is (RQ2.1)*

- Does the participant know about mutation testing?
- If the participant answers by asking: "What do you mean by mutation testing?" Then: Explain briefly what mutation testing is by explaining the core concepts, do not mention how it could be applied in a development workflow and risk introducing bias into the participants later answers. Explain: Mutation testing is a method of assessing the strength of the test of a project by assessing their sensitivity to small changes in the code. In mutation testing, a tool is used to generate altered versions of the code, called mutants. For each mutant, the original tests for the code are then executed. If all tests pass, then the

mutant survived. The mutant is killed if the tests fail. If a large portion of the mutations die, then it is likely that the tests are sensitive to small changes and if a large percentage of the mutants survive, then the tests are likely to be insensitive. A mutation score is calculated based on the percentage of mutants killed:  $\frac{\text{Number of Killed Mutants}}{\text{Total Number of Mutants}}$ .

### *Set of Questions 2: Usage of mutation testing (RQ2.1, RQ2.3)*

- Does Zenseact integrate any mutation testing or similar practices? Can you describe it?
- Skip these questions if mutation testing is not being used.
- Could you describe the process of mutation testing that you currently have in your team, project or company?
- In what way is it being used within the development and testing process?
- What tools are being used to implement the mutation testing?
- How long have you used mutation testing?
- What impact has mutation had on the development process? Has it changed how development or testing is conducted?
- What challenges and problems were there in integrating mutation into the existing development process?
- Was there any positive or negative effects on the development process as a result of applying mutation testing?

### *Set of Questions 3: Prior experience with mutation testing (RQ2.1, RQ2.3)*

- Does the participant have any prior experience with integrating any mutation testing or similar practices? Can you describe it?
- Skip the rest if no experience.
- In what way was it being used within the development and testing process?
- Can the participant say what tools were being used to implement the mutation testing?
- Can the participant explain where mutation testing was integrated?
- What impact did mutation testing have on the development process? Did it change how development or testing was conducted?
- What challenges and problems were there in integrating mutation into the existing development process?

- Was there any positive or negative effects on the development process as a result of applying mutation testing?

*Set of Questions 4: Integrating mutation testing into the workflow (RQ2.1)*

- Do you think that mutation testing could be used to improve the process of development and testing? If so, how should it be integrated into the development and testing process?
- In what stages of the development workflow do they think mutation testing could provide value?
- What tools could be used to validate that a mutated version of the code is worth analyzing? For example could the mutation be run through jobs in the development pipeline to see if another tool would have caught the changed caused by the mutation.
- Other than a mutation score, are there other aspects of mutation that could be informative as part of the development and testing process?
- Would it be useful to get a report on the mutation score as part of test execution during the continuous integration process?
- How much time would you be willing to wait for calculating the mutation score for test cases?
- How often do you foresee wanting to attain a mutation score? On every check-in, daily, weekly, monthly, rarely?
- How much time would you be willing to spend analyzing mutation testing results?
- Could mutation testing help out in development or testing activities other than calculating a mutation score for test cases, such as during the code review process? If so, which activities and how?
- Could historical statistical data on the mutation score for the test suite as it (and the project) evolves over time be of interest?
- (If the subject got experience with mutation testing) Do you limit what operators you use for mutation testing? Could it be worth it to limit the amount of operators to speed up the mutation testing?
- Could mutation testing have any negative effect on the development and testing process?
- How much extra work do you think developers would be willing to spend on adapting projects to work with mutation testing tools? For example ensuring that the unit tests under test can run in a single executable or that the test

framework reports its result in a specific way.

## **Section 3 - Wrap-up and questions to the interviewer**

Asks if the participant has any questions for the interviewer. Interviewer thanks the participant and finishes the interview.

# D

## Interview Data

### D.1 Sub-theme overview

**Table D.1:** Overview of generated sub-themes from the interviews.

Theme	Sub-theme	Description
Test quality	Verify test quality	Verification of the test quality.
	Test quality maintenance	Improving quality of existing tests.
	Mutation score as coverage metric	Mutation score as a coverage metric to see the score of the different parts of the project.
	Mutation score as acceptance criteria	Mutation score as a threshold for accepting new changes.
Prioritization	Prioritize important code	Prioritize mutation testing on important code.
	Task prioritization	Other tasks might be more important than mutation testing.
Risks	Developer motivation	Effects on developer motivation by using mutation testing.
	Focus on the wrong thing	Risk of focusing on the wrong when using mutation testing.
	Diminishing returns	Applying mutation testing on the same code could have diminishing returns.
Implementation	Ease of use	Should be easy to use the mutation tool.
	Initial effort	Initial effort of setting up the mutation tool.
	Developer feedback	Mutation feedback to the developer.
	Time-aware feedback	Time-aware feedback to the developer.
	Mutation operators	Mutation operators to use.
When to use	Code review and pair programming	Use mutation testing result as a code review checkpoint.
	Run periodically	Mutation testing does not have to run all the time, it can be delayed until it won't block other tasks.
	Outside of CI	Using mutation testing as a tool outside of CI.
	Optional step	Optional mutation test usage for developers.