Mutation Testing in Continuous Integration: An Exploratory Industrial Case Study

Jonathan Örgård, Gregory Gay, Francisco Gomes de Oliveira Neto
Chalmers | University of Gothenburg
Gothenburg, Sweden
orgardj@student.chalmers.se, greg@greggay.com, francisco.gomes@cse.gu.se

Kim Viggedal
Zenseact
Gothenburg, Sweden
kim.viggedal@zenseact.com

***Abstract*—Despite its potential quality benefits, the cost of mutation testing and the immaturity of mutation tools for many languages have led to a lack of adoption in industrial software development. In an exploratory case study at Zenseact—a company in the automotive domain—we have explored how mutation testing could be effectively applied in a typical Continuous Integration-based workflow. We evaluated the capabilities of C++ mutation tools, and demonstrate their use in GitHub Actions-based CI workflows. Our investigation reveals that Dextool and Mull could be used in a CI workflow. Additionally, we conducted an interview study to understand how developers would use mutation testing in their CI workflows. Based on our qualitative analysis and practices proposed in literature, we discuss recommendations to integrate mutation testing in a CI workflow. For instance, visualising trends in the mutation score enable practitioners to understand how test quality is evolving. Moreover, tools should have a balance between offering fast feedback and keeping or flagging relevant mutants. Lastly, practitioners raised the need that the mutation should be applied at commit level, and that developers inexperienced with mutation testing should be trained in the implications of the practice.**

***Index Terms*—Mutation Testing, Continuous Integration, C++**

I. Introduction

Structural coverage criteria have been an important source of evidence that code is thoroughly tested in many domains, including safety-critical domains such as the automotive industry, because they offer clear checklists of testing goals that can be objectively evaluated and automatically measured [1]. For example, branch coverage mandates that all outcomes of control-diverting statements, such as if-statements, be executed by test cases. A challenge with traditional coverage metrics is that they mandate that code be executed, but only lightly constrain *how* that code is executed [1]. The resulting tests may be insensitive to subtle problems in the code.

An alternative approach that has gained attention is mutation testing [2], where many small code changes, named "mutants", are seeded into the code, and tests are executed to determine which mutants are detected ("killed"). Mutation operators seed simple syntactic adjustments—e.g., changing $<$ to $\leq$—by identifying appropriate patterns in the code. The proportion of detected mutants is known as the mutation score. This score can be used to indicate the thoroughness of testing [3]. If the test suite can kill a large proportion of the mutants, then it is likely that the tests are *sensitive* to small changes to the code. If a large percentage of the mutants survive, then the test suite should be augmented with additional or improved test cases.

An open question is how best to integrate mutation testing into a typical industrial testing process. For example, developers often test code as part of Continuous Integration (CI) workflows that executes when code is committed to a repository. Mutation testing could be applied either locally or as part of a CI workflow to assess the adequacy of a test suite. In either case, it could be applied across the code base, or selectively to code elements that have been changed in the commit. It is not clear how developers can most effectively apply mutation testing. In particular, two issues hinder widespread industrial adaptation of mutation testing. The first is that mutation testing is notoriously expensive [4]–[6]. Code is expected to be built, tested, and packaged within reasonable time limits so that the developer can get rapid feedback. Mutation testing generally requires tests to be re-executed for each mutant, incurring significant machine cost. If a mutant survives, a developer must examine both the mutant and the test suite, incurring further human cost. The second challenge is that mutation testing tools must operate on a variety of languages—even within a single project. Mutation testing research has largely focused on Java (e.g., PIT [7] and MuJava [8]) to the relatively exclusion of even common languages such as C++. Tools may not even exist for some languages. In others, they may not be sufficiently mature, offer a sufficient variety of mutation operators, or be suitable for use in a automated or CI workflow.

This study focuses on mutation testing as part of CI from the perspective of industrial software development. We conducted this exploratory case study at Zenseact, a company in the automotive domain focusing on autonomous vehicle software. While previous research has been done on mutation in industrial settings (e.g., at Google [5], [6] and Facebook [4]) these studies relied on proprietary software, making it difficult to fully implement their suggestions. In addition, Zenseact uses C++ as its primary development language. Therefore, we focus on existing open-source tools for C++ mutation testing.

We evaluated the capabilities of five C++ mutation tools. We found that Dextool [9] and Mull [10] are potentially suitable for application in a CI workflow, with both offering benefits and drawbacks. To illustrate the use of these tools in CI, we have implemented proof-of-concept demonstrations

using GitHub Actions workflows. In addition, we performed an interview study at Zenseact to explore how developers feel mutation testing should be applied in the testing process. Based on the developers' views of effective mutation testing and applicable features proposed by research and tool authors, some our recommendations discuss how to use mutation score as a coverage and acceptance criteria, as well as creating a balance between efficient and effective mutation during feedback loops in CI. Lastly, offering different options for mutations (e.g., focusing on commits, or choosing subsets of operators) can allow the developer to receive specific feedback to aid test maintenance. In short, this study contributes to knowledge of how to apply mutation testing as part of a developer's workflow, previously identified as an outstanding challenge in the maturation of mutation testing as a practice [2].

II. Background and Related Work

Mutation Testing judges the quality of a test suite by evaluating the ability of the tests to detect subtle changes in the code [11]. A tool generates altered versions of the code, called mutants [12], containing subtle changes based on common syntactic mistakes made by programmers. Mutation operators introduce those syntactic changes to the original code based on grammatical patterns such as changing arithmetical or conditional operators. Mutants are introduced with the intent that they are syntactically valid and semantically useful. Effective mutants will compile ("valid"), and will be "useful" for improving the test suite [13]. Mutants that are not useful include those with equivalent behavior to the original code [14], mutants that trivially cause tests to fail, and mutants whose detecting tests would add no value to the suite (e.g., modifying a statement printing debugging information).

Continuous Integration (CI) is an automated process that executes when code changes are pushed to a repository [15]. A workflow (e.g., GitHub Actions) defines a set of scripted tasks that take place once triggered by the repository update, such as compilation, test execution, and packaging. Instructions for how to perform tasks are defined in build scripts, e.g., a Makefile for C or C++ projects [16]. The intended use of CI is to offer rapid feedback to developers. If compilation or testing fails, code can be rolled back so that other developers are not blocked by the failure. Feedback is often provided from the results of test execution, but other analyses can also be incorporated (e.g., code style checks).

An outstanding problem to solve is how to integrate mutation testing into the development process [2]. Experimental integrations have occurred (e.g., [4]–[6]). However, more observations are needed to make clear recommendations. Our study contributes to this need, and in contrast to past studies, focuses on open-source tools in CI—enabling wider application—and the C++ language—a language that has not been overlooked in mutation research. While tools have been proposed for C++ (including those in our study) [17], we perform the first comparison of the tools.

A common theme in mutation testing research—and in attempts to apply it in industry—is cost, both in human and computational effort. The quantity of mutants generated can cause make mutation testing infeasible at scale [4], [6], [18] However, Google [5], [6] and Facebook [4] have applied mutation testing. Ramler et al. also reported an application at a company developing safety-critical systems [18]. Google only applies mutation to new lines of code that are covered by tests and only generates one mutant per line. They present a limited number of mutations to developers to prevent fatigue. The number of tests executed is restricted to a minimal set. They also apply rules for which code structures are used to generate mutants (e.g., omitting logging code) applying only simple operators such as replacing arithmetic, conditional or relational operators. They further limit operators based on historical data. In turn, Facebook uses machine learning to extract mutation operators from real faults, with the hypothesis that these mutants will be more useful to testers than those generated using traditional operators. Similar approaches have been studied in other research [19].

There has been research on improving the speed of mutation testing. For example, mutation schema encode multiple mutants into one program, in contrast to the standard approach, where each mutant contains a single change [20]. Parts of the code where a mutant could be inserted are replaced with a meta-procedure that determines which mutation to apply. Instead of recompiling between every execution, the program can instead be compiled once and be instantiated to function as any of the mutants. Usaola et al., proposed only executing test cases that reach mutated statements, and using a loop counter to stop execution of mutations with infinite loops [21]. Offut et al., proposed that selection of a subset of mutation operators relevant to a SUT can be almost as effective as using all operators [22]. In addition, Ma et al., propose use of machine learning to predict the operators most likely to produce useful mutants for lines changed in a commit [23].

III. Research Method

We investigate the following research questions:

- **RQ1:** What are the capabilities of existing C++ mutation testing tools, and are any of the existing tools appropriate for use in continuous integration?
- **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 workflow?
  - **RQ2.2:** How can mutation tools be applied to meet the goals of developers?
  - **RQ2.3:** What guidelines should be applied for the use of mutation testing in CI?

We answer these questions through an exploratory case study conducted at Zenseact. Figure 1 gives an overview of our steps to answer each RQ. We have:

1) **Evaluated tools (RQ1):** We evaluated mutation tools for C++ on their capabilities (e.g., mutation operators, generation speed) and applicability in a CI workflow. We also built a proof of concept integration of C++ mutation tools into CI workflows based on GitHub Actions.

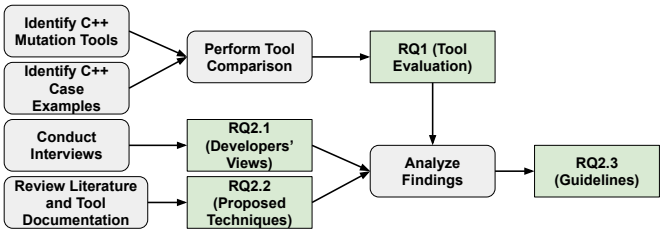

Fig. 1: Overview of the case study process.

2) **Interviewed developers (RQ2.1):** We conducted semi-structured interviews to explore how developers feel mutation testing can be used effectively.
3) **Identified potential solutions (RQ2.2):** We performed a lightweight literature review and analyzed tool documentation to identify techniques or practices that potentially meet developers' needs.
4) **Developed guidelines (RQ2.3):** We synthesise the results of **RQ2.1** and **RQ2.2** to identify mutation guidelines.

A. Case Study Context

Zenseact uses CI to build code and execute test cases. When new code is pushed to the repository, a process is started that involves both automatic and manual actions. The automated actions (jobs) 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. Build and test jobs are skipped if the affected code has not changed since the last execution of the job. The results from the jobs are reported to relevant parties. Failure in early stages will block a merge of new code, and a manual code review is needed before the final merge can be done. Later stages, containing the most expensive jobs, are run after the merge to catch additional problems. Most jobs are run during the day, but some are run at night to avoid blocking other jobs.

Test quality is assessed through manual code review. However, code coverage is measured to ensure that test cases execute all code. Mutation could be used at different points in this pipeline, with varying results. For example, results of other jobs could be used to determine which mutants to use, e.g., static analyses could be used to suggest code to mutate or to filter mutation operators that may not add value.

B. Evaluation of C++ Mutation Tools (RQ1)

We identified five mutation tools. Table I shows a list of tools and versions used[1]. In general, mutation tools work in three phases: code analysis to determine where mutants can be inserted, mutant generation, and test execution to calculate the mutation score [24]. We detail each tool below.

CCmutator is intended to mutate concurrency constructs in multi-threaded applications, and is based on the Clang/LLVM compiler framework [26]. CCmutator generates mutants, but does not handle test execution itself. The tool applies one mutation operator at a time on a given file and allows the user to specify which statements to mutate.

TABLE I: C++ mutation tools examined.

Name	Version	Updated	Source
Dextool	4.1.0-4-g2b5bc097	25 Mar 2022	[9]
Mull	0.17.1	17 Mar 2022	[10]
MuCPP	N/A	7 Jan 2021	[24]
Mutate++	bb341d7 (commit)	25 Nov 2020	[25]
CCmutator	66eca5c (commit)	27 Sep 2013	[26]

Dextool stores mutants and testing results in a SQL database file, which can be reused in future runs. Mutants are only used during test execution if a file has changed. The tool has partial support for mutant schema, where multiple (but not all) mutants are embedded in a single file [20]. However, the schema do not always compile. In such situations, mutants are compiled and tested individually [9].

MuCPP uses the Clang API to find mutation points [24]. It requires a Makefile to function that includes rules to clean up, compile a mutant and test suite, and execute testing. Any testing framework can be used, as long as it can report the mutation testing result in a specific format. The tool uses Git to store mutants in branches. If the existing project uses Git, it must be extracted from its own repository to use MuCPP. The mutation operators to be used can be specified in a config file, and specific files to mutate can be specified.

Mull is a Clang compiler plugin [10], [27] that can be enabled while compiling code to generate mutants. Mull operates on LLVM bitcode, and mutates the program in memory. Mull makes use of mutation schemas, where all mutants are injected into the program while it is being built. Mutants are enabled during execution using conditional flags, activating desired mutations without re-compilation. Mull can also run mutation testing in separate sub-processes, allowing mutations to be tested simultaneously in isolation. A report is then generated to display the result of mutation testing. A configuration file specifies which files to mutate and mutation operators to use. A limitation of Mull is that it can only execute a single test binary, so all test cases must be in a single file.

Mutate++ is a GUI-based web application that runs locally [25]. The user manually uploads files to mutate. The version of the tool used in this study is not mature, and mutants often failed compilation.

To understand the tools, they were applied to a set of C++ projects on a computer with an Intel Core i9-10885H processor, 32GB RAM, and Ubuntu 20.04[2]. MuCPP and CCmutator were installed in a Docker container using the Ubuntu 20.04 image because they had dependencies on a version of Clang incompatible with the remaining tools[3].

[1]MuCPP does not have a version number or changelog.

[2]The settings used for Dextool and Mull can be found at https://github.com/Orgardj/mutation_testing.

[3]Docker has a negligible overhead on I/O operations and CPU performance. Therefore, use within Docker should not affect our observations.

TABLE II: C++ projects used, with size, GitHub stars, and commit. Only C and C++ files are counted.

Name	LOC	Files	Commit ID	Updated	Stars
Corrade	15 359	141	3643585	30 Jan 2022	400
FMT	42 229	65	afbcf1e8	8 Feb 2022	13900
JSON	60 366	104	eec79d4	30 Jan 2022	29300
TimSort	1 584	8	e782512	30 Jan 2022	300
TinyXML-2	5 581	3	a977397	4 Sep 2021	4000
yaml-cpp	18 210	54	edadfec	17 Feb 2022	3200

The tools were applied to a set of six C++ projects of varying size. These projects were either used in previous research [24], [28], or were chosen from 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, and had to have a unit test suite. Table II gives an overview of the projects used, including the size in lines of code (LOC), C/C++ files, commit version, and stars on GitHub (rounded up)—as a way to gauge popularity.

To compare the tools, we gathered data from both inspection of tool documentation, and by executing the tools on the projects described above. The collected data includes:

- Whether the tool could be installed and can generate mutations for each project.
- Whether the tool can be used within CI, and what features each tool offers for customization within a CI environment (e.g., the ability to selectively mutate files or to choose the mutation operators applied).
- Which mutation operators are offered by each tool.
- How many mutants are generated for each project.
- The time to generate mutants.
- The time to execute mutation testing.
- The mutation score attained by test suites.

Using the collected data, we could compare functionality of the tools, as well as their speed and capability at different tasks. This gave us an overview of the capabilities of the tools, including their limitations and applicability within CI. To illustrate the application of mutation testing for C++ in CI, we created two example workflows for the TinyXML-2 project using GitHub Actions[4]. 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).

C. Usage of Tools in CI (RQ2)

We conducted seven semi-structured interviews at Zenseact to understand how stakeholders feel mutation testing could best help them in their practice (RQ2.1). The stakeholders were a mix of software developers, architects, and managers. An overview of the participants is shown in Table III[5].

[4]https://github.com/Orgardj/tinyxml2

[5]Zenseact was founded in 2020. Experience totals include employment at predecessor Zenuity.

TABLE III: Overview of interview participants.

ID	Role	Time at Zenseact	Mutation Experience
P1	Software developer	4 yr, 9 mn	Using it actively at work.
P2	Software developer	1 yr, 6 mn	Read about it.
P3	Cloud architect	6 mn	Academic experience.
P4	SCRUM master and software developer	1 yr, 11 mn	Read about it.
P5	Software developer	4 yr, 9 mn	Read about it.
P6	System architect	2 yr, 4 mn	Read about it.
P7	Chief safety manager	4 yr, 10 mn	Seen others use it at work.

A short introduction was offered 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[6]. Participants were asked, among other things, about when and how mutation testing should be applied in the testing process, the usefulness of mutation score data over time, adverse effects of mutation testing, and how much time should be spent integrating mutation testing or analyzing mutation results. If the participant had prior experience applying mutation testing, we also asked about this. The semi-structured format allowed stakeholders to present ideas.

Thematic analysis was used to analyze the qualitative data from the interviews. Meaningful or expressive pieces of data were identified from the data, called codes. A group of authors coded one interview independently and compared results. The group extracted a very similar set of codes, hence indicating agreement among the reviewers. The remaining coding was done by the first authors. Codes were then given labels summarizing their topic. Those labels were also discussed in two meetings between all authors to ensure reliability of the analysis. The codes were organized into themes and the analysis stopped after three iterations where we notice saturation of the themes.

We conducted a literature review and analyzed tool documentation to identify whether—and how—mutation testing could be applied to effectively and efficiently meet the goals of developers, particularly within a CI/CD context (RQ 2.2). This was intended to be a lightweight review of past research to provide inspiration, not an exhaustive review. The literature review was conducted by searching IEEE Xplore for mutation testing articles. Multiple search strings were applied, combining "mutation testing" with "C++", "continuous integration", "practice", "industry", "at scale", "case study", "cost reduction", and "LLVM". Backward snowballing was also used to identify additional articles. The retrieved articles were filtered for relevancy by first screening the article's title and abstract for relation to our research topic and questions. The title or abstract needed to relate to using mutation testing in industry, optimizing mutation testing, mutation testing in CI, or a mutation tool. After an article was deemed relevant, the article's content was coded and labeled. The same process was applied to the documentation of the tools analyzed in RQ1.

[6]The interview guide can be found in Appendix C in our supplementary material: https://figshare.com/s/e3765a4237759e0e5a16.

TABLE IV: Summary of each tool's capabilities that are relevant to CI integration.

Features	Dextool	Mull	MuCPP	Mutate++	CCmutator
Command line interface.	X	X	X		X
External usable mutants.	X	X	X		X
Execute mutation testing.	X	X	X	X	
Select mutation operator subset.	X	X	X		X
Specify lines of code to mutate.				X	X
Only mutate changed code.	X				
Only mutate code/execute mutants covered by test suite.	X	X			
Stop executing after a specific number of mutants remain live.	X				
Skip previously-killed mutants if no relevant code changed.	X				
Detect redundant tests that do not uniquely kill mutants.	X				
Resume after interruption.	X			X	
Supports mutation schema.	X	X			
Can parallelize mutation testing.	X	X			
Can flag mutants to ignore.	X			X	
Can specify timeout.	X	X	X	X	
Tool updated within two years.	X	X	X	X	
Num. of Traditional Operators	9	10	9	8	1
Could install/compile tool.	X	X	X	X	X
Could install/compile by following installation guide.	X	X	X		
Could generate mutants without modifying project (except compile commands).	X	X	X	X	
Could perform mutation testing.	X	X	X		

Lastly, we answer RQ2.3 by identifying guidelines for mutation testing in industrial development from the combined results of all RQ. The guidelines are based on the developers' goals at Zenseact, but can still be informative for developers at other companies. The techniques and practices identified in RQ2.2 were matched with the themes and sub-themes from RQ2.1 to see which of the techniques and practices could help achieve the developers' goals. This took place as part of the inductive process of thematic mapping. The practices were connected with the themes and sub-themes. For example, the practices *timeout tests* and *selective mutation selection* matched with the sub-theme *time-aware feedback*.

IV. Results and Discussion

A. Evaluation of C++ Mutation Tools (RQ1)

Table IV compares functionality of the five tools that could be of potential benefit in a CI workflow. For example, CI requires a command-line interface, or we may wish to only mutate code that has been changed in the latest commit. The features of the different tools differed substantially. Dextool seemingly offered the most features relevant to CI, with Mull also offering a reasonable subset. The other tools have limitations that make them less effective in CI. MuCPP lacks features related to specifying a subset of code to mutate or for skipping mutants not covered by tests. Mutate++ relies on a GUI. CCmutator cannot execute test cases on mutants itself.

TABLE V: Number of mutants per project, and average time to generate a mutant. Mull may under-report mutant quantity if template functions present (*). MuCPP requires a Makefile.

Project	LOC	Dextool	Mull	MuCPP
Corrade	15359	9573 (4.12e-03s)	2314 (3.11e-02s)	—
FMT	42229	9815 (6.71e-03s)	1998* (0.52s)	—
JSON	60366	6407 (7.01e-03s)	528* (1.09s)	9266 (5.86e-02s)
TimSort	1584	1073 (3.39e-02s)	279 (0.34s)	—
TinyXML-2	5581	1907 (1.94e-03s)	698 (3.00e-03s)	3158 (1.87e-02s)
yaml-cpp	18210	6235 (4.80e-03s)	2147 (2.93e-02s)	—

The tools offered different mutation operators[7]. Four of the five tools offered a subset of traditional mutation operators from literature, albeit with different implementations. CCmutator is intended for multi-threaded C++ programs [26] and has support for 38 specialized operators (as well as the traditional statement deletion operator). MuCPP has the largest variety of supported operators, with 30 class-level operators in addition to traditional operators.

Table IV summarises the outcomes of our attempts to compile, install and run the mutation tools. We experienced the following limitations. Certain mutation operators in Mull had to be disabled. Mutate++ specified incompatible versions of certain Python libraries, and two tools lacked correct or sufficiently detailed installation guides. We were not able to perform mutation testing using Mutate++ and CCmutator because both tools raised errors during, respectively, the test execution and the mutation generation. Therefore, We chose not to proceed further with either tool.

Table V presents the number of mutants generated for each project and the average time to generate each mutant for each tool[8]. MuCPP generates the most mutants, because it supports the most operators. However, it could not be applied to all examined projects. MuCPP requires a Makefile to function, hence hindering its application in projects without their own Makefile. Dextool had the fastest generation time, while still yielding a larger number of mutants than Mull.

After the results from installing and running the tools, *we decided to proceed RQ2 only with Dextool and Mull, as they could be applied to all projects without modification.* Table VI indicates the time spent on mutation testing and the resulting mutation score. Note that Mull can only accept one "test binary", i.e., a file containing compiled test cases, at one time. Therefore, we execute the same test binary for Dextool as well.

Mull is significantly faster in some scenarios, as it does not have to recompile the entire project between test executions. In many cases, compilation is a major contributor to the time of mutation testing. For example—for the TimSort project—Mull executed mutation testing in 2.25 seconds, while

[7]Operators are explained in our supplementary material (Appendix A).

[8]Note that Mull under-reports the number of actual mutants when template functions are present. Therefore, the average generation time is likely lower than reported for Mull in the JSON and FMT projects.

TABLE VI: Mutation execution results. Only one test binary could be executed at a time due to limitations in Mull. Baseline represents execution time of the test suite.

Project	Test Suite	Baseline	Dextool		Mull	
			Time	Score	Time	Score
Corrade	MainTest	0.1s	47009.9s	2.35%	1.5s	5.23%
FMT	core-test	0.1s	66364.6s	2.30%	1.0s	1.80%
JSON	unit-algorithms	0.1s	12980.1s	2.32%	10.1s	9.00%
TimSort	cxx_98_tests	0.4s	5398.9s	59.30%	2.3s	1.43%
TinyXML-2	xmltest	0.1s	2146.1s	77.00%	368.4s	82.00%
yaml-cpp	yaml-cpp-tests	0.8s	29208.0s	72.90%	7254.5s	75.41%

Dextool took more than one hour to execute. TinyXML-2 and Yaml-cpp had a single test binary that included all project tests. Here, we can see that Mull is also faster, but not to the same magnitude as in the other projects.

Clang dependencies: Similarly to any entry in a toolchain, mutation tools have their own dependencies. Such dependencies may be incompatible with the project-under-test or another element of the toolchain. A notable dependency is Clang, a compiler front-end for C/C++. During our study, Dextool was compatible with most versions of Clang (v4.0–13.0), followed by Mull (v9.0–12.0). In turn, CCmutator was only compatible with version Clang 3.2.

Our original intent was to integrate mutation directly into the CI workflow at Zenseact. However, we could not integrate any of the examined tools, as Zenseact relies on a newer version of Clang that are not compatible with any of the tools. Compatibility with Clang was an important revelation, as it is easy to ignore the maintenance burden of introducing a new tool. The project and mutation tool—as well as other components of the CI workflow—may need to be in lockstep on their dependencies. Unfortunately, many mutation tools are developed as part of research projects and are abandoned after the project ends. While a separate toolchain could be maintained to support the older compiler version required to run the tool, the project would also have to be compatible with multiple compiler versions.

Evaluating tool suitability: We were unable to generate mutants using CCmutator and Mutate++. In addition, CCmutator has not been updated in nearly 10 years, and Mutate++ relies on a graphical interface. While MuCPP supports many mutation operators, it requires a Makefile following specific rules and test execution that reports results in a specific format. The tool also has no changelog, making it hard to judge update frequency, and it is unclear what license it has been released under—potentially preventing use in a commercial setting. Therefore, out of the evaluated tools, only two seemed suitable for integration into CI: Dextool and Mull.

Both tools have benefits and drawbacks. Dextool offers a wider variety of options for customizing how mutation testing is performed and selectively mutating code. It is able to generate a greater variety and quantity of mutants than Mull,

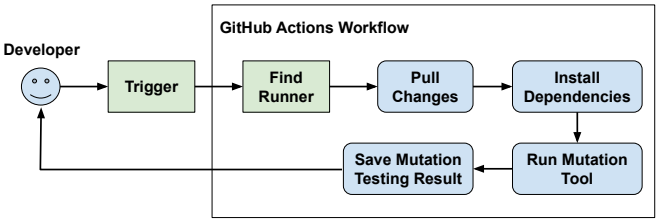

Fig. 2: Generic mutation testing CI workflow. Rectangles represent actions performed by the back-end, rounded rectangles represent actions performed by the tester.

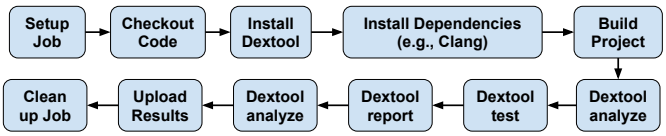

Fig. 3: Overview of the steps for the Dextool CI job.

and can execute the project's full test suite in one invocation. Dextool is also the most mature tool examined, is updated regularly, and had the fewest observed defects.

Mull is not updated as regularly, and its documentation was out of date when examined. However, Mull's support for mutation schemas mitigates the need to re-compile the project for each mutant, such that it can offer faster feedback than Dextool when performing mutation testing on the full project[9]. Therefore, it should remain in consideration when exploring integration of mutation testing into a C++ project.

CI workflow proof-of-concept: We created two CI workflows for the TinyXML-2 project using GitHub Actions as proof-of-concept for the use of mutation testing for C++ in CI. An overview of a generic workflow is shown in Figure 2. After being triggered, GitHub finds a suitable runner for the jobs in the workflow. The worker pulls the relevant commit, then installs all dependencies (along with the mutation tool). The mutation tool(s) are executed on relevant portions of the project. The result of mutation testing is then stored.

The first workflow is triggered on a schedule and uses Dextool or Mull, depending on the configuration, to perform mutation testing on the entire project. The second is triggered by pushes to the repository and uses Dextool to mutate the `git diff` introduced by the push. The workflows are run using the GitHub Docker runner image based on Ubuntu 20.04. Figure 3 shows the concrete actions taken by the worker in the Dextool job.

RQ1 (Tool Evaluation): Dextool and Mull are suitable for application in a CI workflow. Dextool is the most mature, offers high configurability, and was able to generate a large variety of mutations. However, Mull's support for mutation schemas enables faster feedback when performing testing with many mutants.

[9]While Dextool offers support for schemas, it generates partial schemas that require re-compilation. In many cases, these partial schemas also failed to compile, requiring the use of individual mutants.

B. Usage of Tools in CI (RQ2)

The purpose of the interviews was to obtain the developers' view of how mutation testing can be used effectively in the testing process. Identified themes and sub-themes are presented in Figure 4 and explained below.

Test Quality: Mutation testing can be used to assess and improve the quality of a test suite.

- **Verify Test Quality:** When developing a test suite, the mutation score can indicate the overall quality of the suite, and live mutants can be used to identify edge cases that have been missed. The mutation score functions as a sanity check on testing efforts.
- **Quality Maintenance:** When modifying or extending code, tests should be updated or added to reflect new behavior. Participants suggested using the mutation score to examine the change over time in suite quality. This is especially important for delivered code, where faults not caught by the test suite could impact the customer. It is not just new or changed code that could benefit. Surviving mutants can be seen as technical debt that needs to be reduced through the addition of test cases.
- **Mutation Score as a Coverage Metric:** Similar to line or branch coverage, the mutation score can be calculated for different parts of the code to indicate coverage. A developer can investigate code with low mutation scores to identify problems with how that code is tested.
- **Mutation Score as an Acceptance Criterion:** The mutation score can be used as an acceptance criterion for new commits to the repository. Using a threshold could encourage developers to maintain and improve the test suite. Commits below a set threshold could be rejected or require motivation for accepting the change without "sufficient" testing. A threshold is not always ideal—enforcing a threshold could result in wasted testing effort for code that does not end up in the final product. It must also be possible to exclude mutants from consideration, e.g., if they are unreachable. Developers should also not be encouraged to "cheat" the threshold (e.g., writing tests centered around specific mutants that add little value to the overall suite). Care must be taken, and the threshold should be updated as the code and suite change.

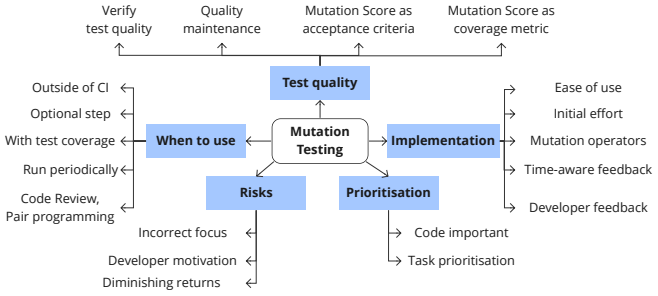

Fig. 4: Themes and sub-themes from our thematic analysis.

RQ2.1 (Summary 1): Mutation testing should be used to verify and maintain quality, and mutation score should be tracked over time.

Prioritization: Mutation testing should be selectively applied, based on either code or task priority.

- **Code Importance:** When first applying mutation testing on a large codebase, there could be many living mutants, making it difficult to know which to focus on. Developers prefer to apply mutation testing selectively to the most important parts of the code before applying it widely. As stated by Participant 3 "*We can't really use mutation testing on a full pipeline that has like 10000 lines of code. Because if we try to use it in that context, then it can be really hard to work with.*". Which code segments are important could be determined by the product domain, or could vary over the lifetime of a project (e.g., dictated by the current development sprint or testing activity). In the context of a company like Zenseact, the most value may come from focusing on safety-critical code.
- **Task Prioritization:** Mutation testing can consume development time both in execution and in test suite improvement. Mutation testing should not block important tasks, such as deploying a critical bug fix.

Risks: There are risks to consider that may affect the effectiveness of mutation testing:

- **Developer Motivation:** If a tool provides false positives or does not work, it will cause interruptions and frustration. Furthermore, results that are not interesting can discourage a developer from further use of mutation testing. For developers to be motivated to use mutation testing, it should work seamlessly and produce productive results. Additionally, the tool should be easy to use, as to not add additional mental fatigue on the developer.
- **Incorrect Focus:** There is a risk that the developer focuses on killing particular mutants without considering the context, potentially missing a bigger problem in the code or tests. If a developer blindly focuses on killing mutants, a test suite could become strictly tied to a particular implementation. This could make it challenging to implement alternative solutions. A developer may over-focus on white-box (code-based) testing—in an effort to increase the mutation score—to the exclusion of tests based on the intended functionality of that code.
- **Diminishing Returns:** Mutation testing is expensive, but that expense is acceptable if the test suite is improved substantially. The value of mutation testing diminishes over time when applied repeatedly to the same code, as developers may not gain any new insight. Consequently, developers should be careful not to over-apply mutation testing to segments of the codebase.

Implementation: The implementation of mutation testing in a developer's workflow will affect the effectiveness with which

developers can use the tool.

- **Ease of Use:** It should be easy for a developer to use mutation testing. As much as possible should be automated—e.g., when tests are added or changed, they should be factored into the mutation score.
- **Initial Effort:** Considerable effort may be needed to get mutation testing to work within a toolchain. PIT, a Java mutation tool, can be easily added to a Maven build script. However, the C++ tools require more effort. Developers unfamiliar with mutation testing would have to be trained in use of the technique.
- **Developer Feedback:** Feedback from execution should show which mutants survived, what operators they represent, why they survived, and what parts of the code are affected. It should be easy for the developer to get an overview and perform an efficient analysis.
- **Time-Aware Feedback:** Developers do not want to wait long for feedback. If used in CI, then mutation testing should take approximately as much time as other jobs in that stage. Even a minute of additional waiting time can be cumbersome. The same advice applies when mutation testing is performed on a developer's local machine.
- **Mutation Operators:** Generating mutants for a subset of operators could save time and offer better feedback. Mutation operators could be prioritized based on the project context or past productivity.

When to Use: Mutation testing could be applied in multiple stages of development or testing.

- **Code Review and Pair Programming:** Mutation testing could be used as part of the code review process. Mutation score could be used as a metric to get an indication of the quality of the test suite or code. Sometimes, pair programming is used to review code continuously. Mutation testing can be used after a pair-programming session to check the quality of created tests and indicate the need for follow-up actions.
- **Run Periodically:** Due to the high machine cost of performing mutation testing, it could delay other jobs, especially on a large codebase. However, not all jobs have to be run on commit. Mutation testing could be run as a post-merge job when there are free computing resources—e.g., at night or on weekends—and results could be sent to the commit author or affected parties.
- **Outside of CI:** CI jobs can be slowed by overhead or limited machine time. 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.
- **Optional Step:** Mutation testing could be an optional step—in CI or locally—for developers or teams who want and have time to improve test quality. The developer would not necessarily have to use mutation testing regularly—it could be used when new tests are added or when there is spare time.

RQ2.1 (Summary 2): Mutation can provide feedback during code review or as a local job executed periodically or with a reduced set of operators. It should be applied selectively based on code and task importance. It must provide rapid and productive feedback, be easy to use, and not require undue setup effort.

Based on a lightweight literature review and comparison of the features of existing C++ mutation tools, we have identified techniques and practices that may enable developers' vision of effective mutation testing. We briefly define the identified techniques and practices in Table VII.

RQ2.2 (Proposed Techniques): Literature and existing tools offer means to decrease the cost of mutation testing. In general, literature focuses on reducing quantity of mutants, while tools focus on increasing speed, e.g., parallelization or skipping tests.

In Figure 5, we map the themes and sub-themes from **RQ2.1** to the techniques and practices from **RQ2.2** to devise guidelines for applying mutation testing in a CI workflow (RQ 2.3). We detail each guidelines below:

Time-Aware Feedback: Developers want feedback as soon as possible. Most of the techniques from **RQ2.2** address this challenge. Techniques that speed mutation testing without decreasing the effectiveness of the practice should be applied to the fullest extent possible, especially those that do not discard mutants (e.g., paralelisation). Mutants will still likely need to be discarded. If so, measures should be taken to ensure productive mutants are kept such as flagging useful mutants to inform selection based on data.

Test Quality Maintenance: Mutation testing can be used to

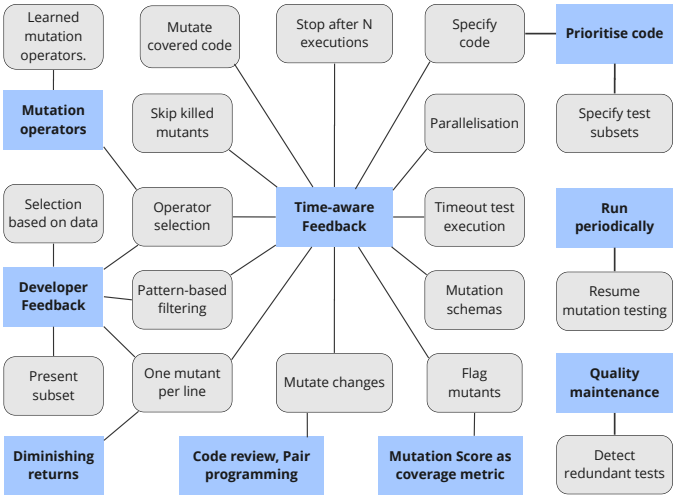

Fig. 5: Mapping techniques/practices (RQ2.2) to sub-themes from RQ2.1 (bold rectangles)

TABLE VII: Mutation techniques and practices from literature and mutation tools.

Technique	Description	Citation	Tool
Operator Selection	A sub-set of effective mutation operators are selected.	[6], [20], [29]	Dextool, Mull, MuCPP, CCmutator
One Mutant Per Line	Only generate one mutant per line of code, as exposure of one mutant in a line often implies exposure of most mutants.	[13]	
Selection Based on Data	Select mutation operators or specific mutations based on past effectiveness for similar statements/the project	[6], [23]	
Learned Mutation Operators	Train a ML model using historical data to create realistic mutants replicating real-world faults from similar code.	[4], [19]	
Present Subset of Mutants	To prevent fatigue, only present a limited subset of live mutants to the developer for inspection (e.g., suggested by metrics).	[6]	
Flag Mutants	Flag mutants as irrelevant, so that they will be ignored in future executions. Useful mutants can also be flagged as helpful, and emphasized in future executions.	[6]	Dextool, Mutate++
Only Run Relevant Tests	Only execute tests that cover mutated lines.	[13]	
Covered Code	Only mutate code that is covered by test cases.	[6]	Dextool, Mull
Specify Tests Subset	Specify subset of tests to execute during mutation testing. Generally also limits mutation to covered lines.	[6]	Dextool, Mull, Mutate++
Specify Code	The user can specify what lines of code to mutate.		Mutate++, CCmutator
Mutate Changes	Only mutate code changed in latest commit.	[13], [23]	Dextool
Pattern-Based Filtering	Mutate or omit code that matches pre-specified patterns (e.g., omitting debugging or logging code).	[6]	
Mutant Schema	Multiple mutants inserted in single file, activated using flags, avoiding re-compilation.	[20], [21]	Dextool, Mull
Live Mutant Threshold	After a specific number of live mutants have been detected, mutation testing stops.		Dextool
Skip Killed Mutants	Mutation result is stored between runs, so killed mutants can be skipped if there has been no change to the code or test suite.		Dextool
Redundant Tests	Redundant tests that do not uniquely kill mutants are flagged for inspection.		Dextool
Resume Mutation Testing	Mutation testing can be resumed after an interruption. Enables pausing, e.g., if machine resources are needed.		Dextool, Mutate++
Parallel Mutation	Parallelization of mutation testing execution.	[6]	Dextool, Mull
Timeout Execution	Avoid infinite loops by stopping test execution based on a specified time bound.		Dextool, Mull, Mutate++
Stop After N Executions	Avoid potential infinite loops by stopping test execution after a line is executed a specific number of times.	[21]	

understand how test quality changes over time and maintain quality, by using mutation scores as a coverage metric. Furthermore, with a large codebase, the essential areas of the code should be prioritized for mutation testing. Mutation testing should be run run periodically when there is free machine time to not block critical jobs.

Mutation Testing at Commit: Mutation testing could be used as a optional or mandatory quality check when code is committed or a pull request is made. This could be as part of a code review process or after a pair programming session. Only the changed code should be mutated, and only tests that cover the code should be executed. The mutation score could be used as a coverage or acceptance metric to judge the quality of tests covering changed code.

Developer Motivation: The mutation tool should be easy to use, work correctly, and provide sufficient and productive feedback. The mutation reduction techniques mentioned for time-aware feedback can, if configured correctly, ensure that the developers get both faster and more productive feedback.

Risks: Inexperience with mutation testing can lead developers to misplace their focus or over-emphasize tests tied to a specific implementation. To avoid these problems, developers should be trained to understand mutation testing and its results. Similarly, the initial setup of the mutation tool should be possible without extensive experience or effort.

> **RQ2.3 (Guidelines):** Our recommendations include, among others, that mutation testing should offer feedback as fast as possible (while taking care not to discard productive mutants), that essential areas of the code should be prioritized for mutation testing, that the mutation score be tracked over time to understand evolving test quality, that mutation be applied to changed code at commit, and that developers inexperienced with mutation testing should be trained in the implications of the practice.

V. Threats to Validity

Regarding the validity of our interview study, the terms related to mutation testing could have different meaning to different interview participants. Therefore, interviewees were offered an introduction to the topic and term definitions. We conducted interviews at a single company, which limits the generalizability of the results. We also acknowledge that the limited number of interviews risks biasing our conclusions. However, we argue that the interview participants have offered sufficiently detailed and consistent responses to capture how mutation testing could be effectively employed, at least, at Zenseact. We also saw connections between our interview data and evidence supported by previous empirical studies (RQ2.3). Therefore, we believe that our findings are applicable to other companies employing a similar CI-based development process.

Lastly, our experiments used open source technologies, and were applied to multiple sample projects, increasing the potential applicability of our findings. Documentation quality differed significantly between the evaluated mutation tools. This hinders understanding of their use and applicability. However, we applied significant effort to document and apply tools consistently. Our proof-of-concept workflows are built on the popular GitHub Actions framework and can be adapted to other projects and version control platforms (e.g., gitlab). We could not integrate the mutation tools directly with Zenseact projects due to limitations in the mutation tools. This limited

our ability to evaluate our guidelines. We aim to implement our guidelines as part of future work.

VI. Conclusion

We evaluated the capabilities of existing C++ mutation tools, demonstrating that Dextool and Mull are potentially suitable for application in a CI workflow. On the other hand, the lack of compatibility with newer versions of clang hindered applicability of those tools at our industry partner. Additionally, we performed a literature study and an interview study at Zenseact to identify suggestions on how mutation testing can best be used within CI. Based on the developers' views of effective mutation testing and the features proposed in literature and existing tools, we discuss many recommendations for using mutation testing in CI. For instance, mutation score can be incorporated into the CI pipeline to offer insights on test adequacy and quality of test maintenance, particularly when looking at trends of the score over various builds. Practitioners should also be strategic when optimising mutation tools for timely feedback. For instance, strategies such as parallelisation or running over downtime (e.g., overnight or weekends) are preferred over those that would discard mutants. In the latter case, flagging useful mutants can be used as data for selective mutation in future builds.

Our observations and recommendations can be used to identify areas of improvement for mutation testing tools or to help developers make effective use of mutation testing in their practice. In future research, we plan to explore the capabilities of mutation tools for other languages, implement missing features into mutation tools, extend our GitHub Actions workflows for additional application scenarios, tools, or languages, and extend our study to additional companies.

References

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

[2] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Mutation testing advances: An analysis and survey," *Advances in Computers*, vol. 112, p. 275–378, 2019.

[3] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[4] M. Beller, C.-P. Wong, J. Bader, A. Scott, M. Machalica, S. Chandra, and E. Meijer, "What it would take to use mutation testing in industry—a study at facebook," *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021.

[5] G. Petrovic, M. Ivankovic, G. Fraser, and R. Just, "Does mutation testing improve testing practices?" *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021.

[6] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Practical mutation testing at scale: A view from google," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3900–3912, 2022.

[7] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: A practical mutation testing tool for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 449–452. [Online]. Available: https://doi.org/10.1145/2931037.2948707

[8] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava: a mutation system for java," in *Proceedings of the 28th International Conference on Software Engineering*, 2006, pp. 827–830.

[9] J. Brännström, "Dextool Mutate," https://github.com/joakim-brannstrom/dextool/tree/master/plugin/mutate.

[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)*, April 2018, pp. 25–31.

[11] R. Demillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, p. 34–41, 1978.

[12] T. Budd, "Mutation analysis of program test data," Ph.D. dissertation, Yale University, 1980.

[13] G. Petrovic, M. Ivankovic, B. Kurtz, P. Ammann, and R. 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.

[14] M. Papadakis, M. Delamaro, and Y. Le Traon, "Mitigating the effects of equivalent mutants with mutant classification strategies," *Science of Computer Programming*, vol. 95, p. 298–319, 2014.

[15] M. Shahin, M. Ali Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.

[16] S. Karus and H. Gall, "A study of language usage evolution in open source software," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 13–22. [Online]. Available: https://doi.org/10.1145/1985441.1985447

[17] A. Denisov and S. Pankevich, "Mull it over: Mutation testing based on llvm," *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2018.

[18] R. Ramler, T. Wetzlmaier, and C. 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.

[19] D. B. Brown, M. Vaughn, B. Liblit, and T. Reps, "The care and feeding of wild-caught mutants," *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[20] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," *Proceedings of the 1993 international symposium on Software testing and analysis - ISSTA '93*, 1993.

[21] P. R. Mateo and M. P. 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.

[22] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, p. 99–118, 1996.

[23] W. Ma, T. Titcheu Chekam, M. Papadakis, and M. Harman, "Mudelta: Delta-oriented mutation testing at commit time," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021.

[24] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, and J. J. Domínguez-Jiménez, "Assessment of class mutation operators for C++ with the mucpp mutation system," *Information and Software Technology*, vol. 81, pp. 169–184, 2017.

[25] nlohmann, "Mutate++," https://github.com/nlohmann/mutate_cpp.

[26] M. Kusano and C. Wang, "Ccmutator: A mutation generator for concurrency constructs in multithreaded c/c applications," *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 722–725, 2013.

[27] A. Denisov and S. Pankevich, "Mull," https://github.com/mull-project/mull/.

[28] M. A. 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)*, 2021, pp. 150–152.

[29] P. Delgado-Pérez, S. Segura, and I. Medina-Bulo, "Assessment of c++ object-oriented mutation operators: A selective mutation approach," *Software Testing, Verification and Reliability*, vol. 27, no. 4-5, 2017.