UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Gaukhar Dauzhan

Mutation Testing for Improving Fault Detection Capability of Unit Tests: A Case Study

Master's Thesis (30 ECTS)

Supervisors: Dietmar Pfahl, PhD
Eerik Potter, MSc

Tartu 2021

Mutation Testing for Improving Fault Detection Capability of Unit Tests: A Case Study

Abstract:
Mutation testing helps assess and improve the quality of a test suite with respect to its fault detection capability. It generates mutants by making small syntactic changes in a program and checking whether tests can detect the changes. The more mutants are detected, the stronger the test suite is considered to be. While mutation testing is generally recognized to be superior to traditional code coverage criteria in terms of evaluating test suite quality, it has some limitations that prevent it from being widely adopted in the industry. These limitations are computational expensiveness and significant developer time spent evaluating mutants and adding tests to kill them. In this study, past software bugs of several Scala systems at Twilio have been used to determine whether mutation testing could have helped to prevent the bugs. In particular, it has been checked whether new tests added during the process of mutation testing could have detected the faults. The results have shown that mutation testing was beneficial only for 1 bug out of 17; the bug was a faulty algorithm with complex logic. It was discovered that mutation testing could bring other benefits, such as finding redundant test cases and improving code quality. However, there exist critical impediments for using mutation testing, such as the immaturity of the testing tool for Scala, high effort to integrate the tool into the development flow, and a large number of equivalent mutants. Overall, the minor benefits that mutation testing could give do not outweigh the effort required to use it at the company effectively.

Keywords:
Mutation testing, fault detection, test suite effectiveness

CERCS: P170 Computer science, numerical analysis, systems, control

Mutatsioontestimine Ühiktestide Rikete Avastamisvõime Parandamiseks: Juhtumiuuring

Lühikokkuvõte:
Mutatsioonitestimine aitab hinnata testikomplektide kvaliteeti ja parandada rikete tuvastamise võimet. See genereerib nn mutante, tehes programmis väikeseid süntaktilisi muudatusi ja kontrollides, kas testid suudavad tehtud muudatusi tuvastada. Mida rohkem mutante tuvastatakse, seda tugevamaks loetakse testipaketti. Ehkki mutatsioonide testimist loetakse testipaketi kvaliteedi hindamisel tavapärastest ühiktestide katvusest paremaks, on sellel mõned piirangud, mis takistavad selle laialdast kasutuselevõttu tarkvara arenduses. Peamisteks piiranguteks on arvutuslik kulukus ja arendajate märkimisväärne aeg, mis kulub mutantide hindamisele ja nende elimineerimisele lisa testidega. Käesolevas töös on uuritud Twilio mitmete Scala teenuste varasemaid tarkvaravigu, et teha kindlaks, kas mutatsioonide testimine oleks võinud vigu ennetada. Eelkõige on

kontrollitud, kas mutatsioonide testimise käigus lisatud uued testid oleks võinud rikkeid tuvastada enne kui hilja. Tulemused on näidanud, et mutatsioonitestimine oli kasulik ainult ühe vea korral 17-st; mille korral oli tegemist keeruka algoritmiga. Avastati, et mutatsioonide testimine võib tuua ka muud kasu, näiteks üleliigsete testijuhtude leidmine ja koodi kvaliteedi parandamine. Samal ajal on mutatsioonitestide kasutamisel ka kriitilisi takistusi, nagu näiteks Scala testimisvahendi ebaküpsus, suured pingutused tööriista integreerimiseks arendusvoogudesse ja suur hulk samaväärseid mutante (palju müra). Seetõttu ei kaalu väiksemad eelised, mida mutatsioonitestimine anda võib, ettevõttes üle selle tõhusaks kasutamiseks vajalikke jõupingutusi.

Võtmesõnad:
Mutatsioonide testimine, rikete tuvastamine, testipaketi efektiivsus

CERCS: P170 Arvutiteadus, arvanaluus, susteemid, kontroll

Contents

1 Introduction **6**

2 Mutation testing **7**
2.1 Mutation testing process 11
2.2 Mutant operators 13
2.3 Mutation testing in different programming languages 14
2.4 Mutant killing conditions 14
2.5 Mutant types 16
2.6 Mutation testing versus code coverage driven testing 16
2.7 Problems and solutions in mutation testing 18

3 Related work **20**
3.1 Industrial case studies 20
3.2 Correlation between mutants and real faults 21
3.3 Correlation between different test coverage criteria and fault detection . 22

4 Methodology **23**
4.1 Mutation testing application plan 23
4.2 Justification for the choice of the mutant set 23
4.3 Experiments with past bugs 24
4.3.1 Locating past bugs 25
4.3.2 Preparing past bugs for experiments 25
4.3.3 Experiment procedures 26
4.4 Subject programs 30
4.5 Mutation testing tool 31
4.5.1 Configuration 32
4.5.2 Mutant operators 32
4.5.3 Mutants and Metrics 33

5 Results **35**
5.1 Service 1: Ratedeck Parser 36
5.2 Service 2: Cost API 39
5.3 Service 3: Provider Service 40

6 Analysis of mutants **41**
6.1 Equivalent and redundant mutants 41
6.2 Non-equivalent and non-redundant mutants 43

7 Discussion **44**
7.1 Answers to the research questions 44
7.2 Identified benefits and barriers 44
7.3 Using mutation testing in industry 45
7.4 Limitations of the study 46

8 Conclusions **47**

References **51**
I. Examples of test cases 52
II. Mutation testing report 55
II. Licence 56

1 Introduction

Mutation testing is a state-of-the-art software testing technique in which small syntactic changes (e.g., replacing "-" with "+") are made in a program to check whether tests can detect them. The changed program is called a *mutant*. If the tests fail to distinguish between the mutant and the original program, it could be concluded that the test suite is not strong enough and should be improved. Thus, mutation testing allows for testing the test suite's quality in terms of fault detection capability.

This study aims to determine whether mutation testing can help improve the fault detection capability of unit tests written by Twilio developers. Studies have shown that mutation testing has a higher correlation with fault detection compared to simple code coverage criteria [35]. However, rarely has work been done that checked that mutation testing would have helped prevent real bugs. Thus, to understand the value of mutation testing for the company, it is essential to determine whether mutants resemble or whether mutants are coupled with real faults produced by Twilio developers.

Mutation testing also has some limitations that prevent it from being widely adopted in the industry, such as high computational cost and developer time required to identify equivalent mutants, add tests to kill mutants, and verify program correctness. Given the limitations, it is crucial to understand how much value mutation testing brings to Twilio systems before introducing it in the development flow.

Thus, the major research question of this study is as follows:

- **RQ**. Does mutation testing improve the fault detection capability of the test suites used by Twilio?

Based on the results, one of the following research questions will be answered:

- **RQ1** If the answer to RQ is "no", what are the reasons why mutation testing does not improve the fault detection capability of the considered test suites?
- **RQ2** If the answer to RQ is "yes", which mutant operators are the most effective in detecting faults?

To answer the research questions, the following experiments will be conducted. Several Twilio projects, which contain faults not detected by the used test suites, will be selected for analysis, and mutation testing will be applied to check whether the faults could be found with the improved test suites.

2 Mutation testing

Mutation testing is a testing technique in which small deviations of an original program are created, one at a time, to see whether a test suite can detect them. The modified versions of the program are called mutants. If the tests detect the mutants, then the mutants are *killed*; otherwise, they *survive*. Based on this, one can calculate a *mutation score*, or mutation coverage, which is the number of killed mutants divided by the total number of mutants. This metric can be used as a test adequacy or test effectiveness criterion. It can help to determine when to stop adding more test cases and whether the existing test suite is comprehensive.

Mutation testing is based on the two fundamental hypotheses: *Competent Programmer Hypothesis* and *Coupling Effect*. Both concepts were proposed by Demillo et al. in 1978 [9]. *Competent Program Hypothesis* suggests that programmers are in general competent, which means that they write programs that are close to the correct versions. Therefore, the errors can be easily corrected via small syntactic changes. *Coupling effect* hypothesis suggests that the test data that can detect minor errors can also distinguish complex errors. In the language of the mutation testing, it means that the complex faults are coupled with simple faults such that the test data that can detect simple faults will also be able to detect more complex faults [10].

There are different ways in which mutation testing is used in practice:

- Evaluate test suite with regards to its effectiveness
- Generate (augment) test suites
- Compare test suites based on their effectiveness
- Minimize test suites
- Localize faults

Let us consider an example where we could demonstrate each of the following ways of using mutation testing.

We have a program that tells us whether a person can receive an age-related pension. To receive that pension, the person must reach the retirement age of 63 years old. The program is written as follows:

```
def canGetPension (p : Person) : Boolean = {
    p.age >= 63
}
```

And the corresponding test suite (a) consists of the following tests:

```
test ("The person has reached retirement age") {
    val person = Person("Askar", 70)
    assert(canGetPension(person))
}

test ("The person has not reached retirement age") {
    val person = Person("Anna", 20)
    assert(!canGetPension(person))
}
```

This piece of code has 100% statement and branch coverage. However, it can be seen that the test for the boundary value is missing, and the mentioned testing criteria do not capture this. If we apply mutation testing, the following mutants will be generated:

```
// Mutant 1:
def canGetPension (p : Person) : Boolean = {
    p.age > 63
}
// Mutant 2:
def canGetPension (p : Person) : Boolean = {
    p.age < 63
}
// Mutant 3:
def canGetPension (p : Person) : Boolean = {
    p.age == 63
}
```

Evaluate test suite with regards to its effectiveness - evaluate the effectiveness of the test suite based on the mutation score.

As a result of mutation testing, the mutation score will be around 66.7%: two out of three mutants will be killed. With Mutant 1, all tests pass (Mutant 1 survived), while with the other two mutants, some of the tests fail (Mutant 2 and Mutant 3 are killed). In particular, Mutant 2 will make both tests fail, and Mutant 3 will make the first test with the description "The person has reached retirement age" fail.

Based on mutation testing, the test effectiveness of this program would correspond to the mutation score of 66.7%.

Generate (augment) test suites - generate test suites or add additional test suites based on survived mutants.

In order to kill the survived mutant, we add the following test that checks the boundary value:

```
test ("The person has reached the minimum retirement age") {
    val person = Person("David", 63)
    assert(canGetPension(person))
}
```

And the whole test suite (b) will now consist of three tests:

```
test ("The person has reached retirement age") {
    val person = Person("Askar", 70)
    assert(canGetPension(person))
}

test ("The person has not reached retirement age") {
    val person = Person("Anna", 20)
    assert(!canGetPension(person))
}

test ("The person has reached the minimum retirement age") {
    val person = Person("David", 63)
    assert(canGetPension(person))
}
```

Thus, using mutation testing, we augmented our test suite. If we rerun mutation testing, the mutation score will be 100%.

Compare test suites - compare different test suites by mutation score; the higher the mutation score, the stronger the test suite is considered to be. If the mutation score is the same for all test suites, the test suite with fewer tests is preferable.

In the considered example, we can compare two test cases - the original test suite with two tests and the enhanced test suite with three tests. Based on the mutation score, we would choose an enhanced test suite (b) with three tests.

Minimize test suites - remove tests that do not affect mutation score. If in the given example the initial test suite (a) had the following test in addition to the original two tests:

```
test ("The person has not reached the minimum retirement age") {
    val person = Person("David", 10)
    assert(canGetPension(person))
}
```

Then using mutation testing, we could notice that this test does not affect mutation score because there would still be one survived and two killed mutants. In other words, this test does not bring any value and therefore could be removed.

Localize faults - locate real faults.

To demonstrate this use case, let us consider a modified version of the original program that would contain an error but the same original test suite.

```
def canGetPension (p : Person) : Boolean = {
    p.age > 63;
}
```

```
test ("The person has reached retirement age") {
    val person = Person("Askar", 70)
    assert(canGetPension(person))
}

test ("The person has not reached retirement age") {
    val person = Person("Anna", 20)
    assert(!canGetPension(person))
}
```

In the given example the program again has 100% statement and branch coverage and all tests still pass. If we apply mutation testing, we can produce the following mutants:

```
// Mutant 1:
def canGetPension (p : Person) : Boolean = {
    p.age >= 63
}
// Mutant 2:
def canGetPension (p : Person) : Boolean = {
    p.age < 63
}
// Mutant 3:
def canGetPension (p : Person) : Boolean = {
    p.age == 63
}
```

When we run our test on the given mutants, the first mutant will survive while the other two will be killed. Now we want to write a missing test that would kill this mutant. As we write the test based on the original requirements, it will be as follows:

```
test ("The person has reached the minimum retirement age") {
    val person = Person("David", 63)
    assert(canGetPension(person))
}
```

When we run the test, the program fails, and we can see an error in the implementation. The correct version of the program is as follows:

```
def canGetPension (p : Person) : Boolean = {
    p.age >= 63
}
```

After correcting the program, we need to conduct mutation testing again, and we would get the same results as earlier with a 100% mutation score. It is important to note that when adding the test, we need to write a proper test following business requirements and not the one that would only kill the mutant. However, this requirement is not specific to mutation testing. Similarly, when writing tests to cover specific pieces of code, we

also need to write the tests that would verify the desired behavior and not simply the ones that would increase coverage percentages.

Other benefits of using mutation testing may include code refactoring and effective debugging [40].

The whole mutation testing process is described in detail in the following section.

2.1 Mutation testing process

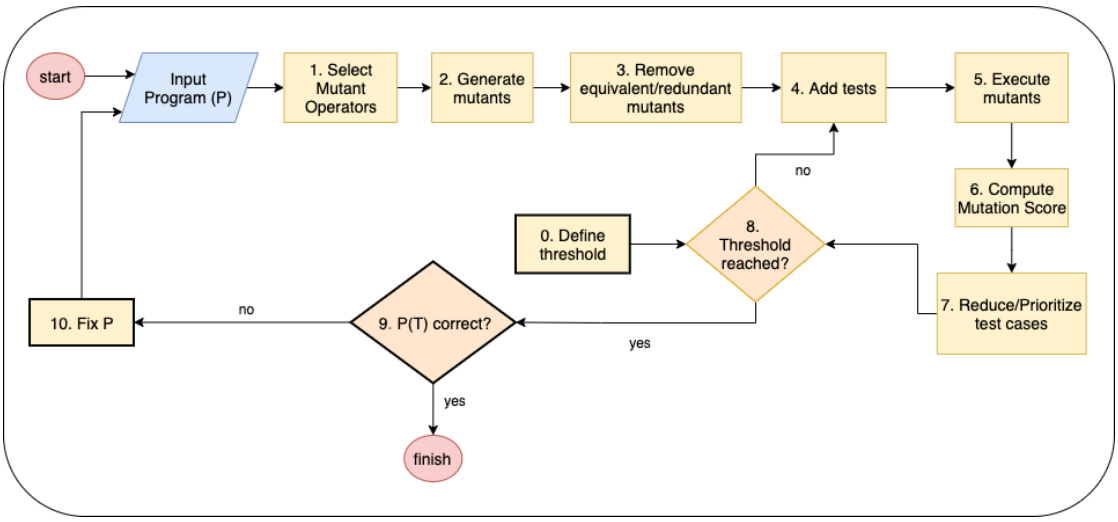

Figure 1. Mutation Testing Process. Adapted from [35]

The general mutation testing process described by Papadakis et al. [35] in the survey of the latest advances in mutation testing is shown in Figure 1. The steps with bold frames require human intervention, while the rest of the steps could be fully or partially automated.

The steps of the mutation testing process may vary depending on the use case, goals, and available tools. For example, instead of trying to achieve a certain threshold for mutation score, one could try to kill all killable, i.e., non-equivalent productive mutants [39, 6] (see Section 2.5 for mutant types). In addition to that, the 7th step about reducing and prioritizing test cases is also optional. It depends on the mutation testing tool and is conducted if the goals of mutation testing include minimizing or optimizing a test suite.

0. Before the process, a threshold for mutation score may be defined. Theoretically, this threshold reflects the level of confidence in a test suite, and 100% would mean the highest confidence level. In practice, however, there is no generally agreed minimum threshold for the mutation score [35]. Studies only suggest that higher mutation scores correlate better with fault revelation. This is mainly because

equivalent and redundant mutants can significantly distort the mutation score, making it harder to define the exact threshold. In industrial projects, mutation score can be impractical also because of the restricted code area that is mutated. Suppose only some lines of code or separate files are mutated, as opposed to the whole codebase. In that case, there is a higher chance that the mutation score will vary significantly and thus be not representative of the adequacy of the test suite. Thus, instead of asking whether the mutation score threshold is reached, a developer can check whether there exist any survived non-equivalent mutants for which the tests should be added.

1. Select mutant operators. For example, a mutant operator could be changing ">" to ">=". In practice, mutant operators depend on the programming language and the mutation testing tool. Section 2.2 describes mutant operators.

2. Generate mutants. This part is usually done using a mutation testing tool.

3. Remove equivalent and redundant mutants. Equivalent mutants are impossible to kill, and redundant mutants get killed together with other mutants. Both types of mutants are generally considered useless and undesirable as they distort the mutation score and take active developer time to be recognized. Evaluating equivalent and redundant mutant could be only partially automated and, for the most part, remains a manual job. Section 2.4 describes them in detail.

4. Add automatically or manually test cases to kill survived mutants.

5. Run the updated test suite against the mutants.

6. Compute the mutation score.

7. Remove ineffective test cases or prioritize them such that the most effective ones are executed first. Ineffective test cases do not change the mutation score, while the most effective test cases kill the highest number of mutants.

8. If the mutation score is above the threshold or no survived non-equivalent mutants are left, go to the next step. Otherwise, return to step 4 of generating test inputs and repeat steps 4-7.

9. Check whether the enhanced test suite passes, i.e., whether the code still does not contain any faults. If the newly added tests fail, a fault has been detected, which was not found by the original test suite. This step is related to the human oracle problem of asserting the program behavior. If the program is correct, the process is completed. Otherwise, go to the next step.

10. Fix the program, return to step 1 and repeat the whole process.

In this study, instead of a threshold for mutation score, step 8 asks, "Are there any survived non-equivalent mutants?" and step 7 of reducing/prioritizing test cases is not included since it is not the goal of the thesis. The mutation testing tool selects all available mutant operators, generates mutants, runs each mutant against the test cases, and computes the mutation score. The developer part includes considering each mutant for equivalency, manually adding test cases, and evaluating program correctness. The exact implemented steps in the experiments of the study are shown and described in Section 4.3.3.

2.2 Mutant operators

In the context of software testing, mutants are small variations of the program with minor syntactic changes. The syntactic changes are made by applying so-called mutant operators that transform the syntax of the program. There is a set of operators proposed by Offutt et al. [31] that is considered a minimum standard set for mutation testing [23]. This set includes the following types of operators:

- Absolute value insertion (e.g. "e" –> "abs(e)")
- Arithmetic operator replacement (e.g. "a+b" –> "a-b")
- Logical connector replacement (e.g. "a or b" –> "true")
- Relational operator replacement (e.g. "a > b" –> "false")
- Unary operator insertion (e.g. "a" -> "!a")

Examples of other operators:

- Statement deletion
- Statement insertion or duplication
- Replacement of variables with other variables of the same type
- Removal of a method body
- Removal of array contents during the array initialization
- Changing constant values (e.g. "var a = 10" –> "var a = 20")
- Changing method expressions (e.g. "filter" –> "filterNot")

The mutant operators differ based on the programming language and mutation testing tool.

2.3 Mutation testing in different programming languages

The types of mutant operators used in experimental studies depend on the programming language's constructs and the programming paradigm. For example, strongly typed languages can have fewer mutant operators than weakly typed languages [20]. Another example is that object-oriented code tends to have simpler methods compared to interactions between the methods; thus, intra-method mutants may be less effective [27]. Moreover, Le et al. [24] argue that functional programming languages may benefit from mutation testing more than imperative programming languages as some features of functional languages can help to understand generated mutants better. The features are code compactness, data flow simplicity, referential transparency, and clean semantics.

Choice of mutation testing tool may also affect the results of mutation testing. It has been shown by several studies concentrated on Java testing tools that the results produced by these tools disagree with each other [21, 14, 22]. Also, depending on the implementation, performance may vary across the tools.

The most popular programming languages for mutation testing are Java and C. Other programming languages for which there exist mutation testing tools include C++, C#, Closure, Crystal, Elixir, Erlang, Go, Haskell, Javascript, PHP, Python, Ruby, Rust, Scala, Swift, and Smalltalk [1].

2.4 Mutant killing conditions

For a mutant to be considered killed by a test suite, the following conditions should be met:

1. Reachability: The mutant must be reached by a test.
2. Infection: Test input data must cause incorrect program state when run on the mutant.
3. Propagation: Incorrect program state must propagate to the observable program state (e.g., method output) and be checked by the test.

These conditions constitute the RIP model for generating test cases [32].

However, a mutant can be considered killed when only some of the described criteria are satisfied.

Consider the following simple program that checks whether the minimum of two values is less than 5:

[1] https://github.com/theofidry/awesome-mutation-testing

```
def minLessThanFive(a: Int, b: Int) : Boolean = {
    val minVal = min(a, b)
    if (minVal < 5) {
        true
    } else {
        false
    }
}
```

Mutating `min` to `max` would result in the following mutant program:

```
def minLessThanFive(a: Int, b: Int) : Boolean = {
    val minVal = max(a, b)
    if (minVal < 5) {
        true
    } else {
        false
    }
}
```

Test case `minLessThanFive(1, 3)` on the mutant would cause incorrect program state (`minVal` = 3) but the same output as the original program, which would be `true`. Thus, this test case would only satisfy the first two killing conditions.

Test case `minLessThanFive(1, 20)` on the mutant would cause incorrect program state (`minVal` = 20) as well as incorrect program output, which would be `true`. Thus, this test case would satisfy all three mutant killing conditions.

Based on the required killing conditions, we can classify mutation testing techniques into the following three groups: *strong*, *firm*, and *weak*.

Strong mutation testing happens when all three conditions are met, i.e., a mutated program gives an output that is different from the original program.

Weak mutation testing takes place when the program state changes immediately after executing the mutant, but this does not necessarily propagate to the output of the program. Thus, it requires only the first and second conditions to be met. The program state can be expressed in the variable reference, arithmetic, boolean, and relational expression. The advantage of weak mutation testing is that it does not require the execution of the whole program and is therefore cheaper.

Firm mutation testing is similar to weak mutation testing, with the only distinction being that the difference in program state is checked later in the program after executing the mutant.

Overall, weak mutation testing is less effective than firm mutation testing, and firm mutation testing is less effective than strong mutation testing.

2.5 Mutant types

If a mutant does not get killed, it means it *survived* or *lived*. A mutant can survive for two reasons: either because the test suite is not thorough enough to detect the error or because the mutant is semantically equivalent to the original program.

Equivalent mutants are syntactically different but semantically equivalent to the original program. Thus, it is not possible to find a test that would kill them. Equivalent mutants can affect the mutation score considerably, as it has been reported that there could be 10 to 40% of equivalent mutants generated for a program [29, 30]. Therefore, equivalent mutants are undesirable and need to be excluded from the analysis.

In addition to equivalent mutants, there can also be *redundant mutants*, mutants that do not bring any value but affect the mutation score because they are killed when other mutants are killed. Redundant mutants can be in the form of *duplicated* mutants or *subsumed* mutants. *Duplicated* mutants are equivalent to each other, but different from the original program [34]. *Subsumed* (also known as *joint*) mutants are killed by the same test suite that kills other mutants [33]

Identification of such undesirable mutants - *equivalent* and *redundant* - is an undecidable problem [34], hence much of the research has been focused on identifying and removing them. However, Petrovic et al. in their study based on the industrial application of mutation testing at Google [40] suggest that this classification is not practical. They argue that there exist equivalent mutants that help developers to reveal issues and refactor the code. At the same time, there exist non-equivalent and non-redundant mutants that are impractical to kill. Thus, developers should not waste time writing redundant tests to kill them. For instance, a mutant that replaces ">" with ">=" when comparing floating-point numbers does not bring any value. Another illustrative example is replacing an exception-related string message with an empty string. Other reported examples include changing string concatenation to string multiplication in Python, changing a network timeout, subtracting from infinity. Petrovic et al. argue that killing these kinds of mutants would require extra human effort that would result in fragile tests that only worsen the test suite [40].

Instead, Petrovic et al. [40] propose a new classification of *productive* and *unproductive* mutants for usage in industry. A *productive* mutant is either a killable mutant that corresponds to an effective test or an equivalent mutant that leads to advanced knowledge or increased code quality. An *unproductive* mutant is either an equivalent mutant that does not bring any other benefits or a non-equivalent mutant writing the tests for which would only decrease the quality of the test suite.

2.6 Mutation testing versus code coverage driven testing

There exist different test criteria other than mutation score that can be used to test the adequacy of the test suite and thus provide greater confidence that software is reliable

and of high quality.

Typically used code coverage types are listed below:

- *Statement coverage*. Statement coverage checks that every statement in the program has been executed.
- *Decision / Branch coverage*. Decision, or branch coverage, checks that every branch has been executed. If there is an *if* statement, it checks that both *true* and *false* branches have been executed.
- *Function coverage*. Function coverage checks that every function in the program has been called.
- *Condition / predicate coverage*. Condition coverage checks that every boolean sub-expression in the conditional statement has been evaluated both to True and False.
- *Path coverage*. Path coverage checks that every possible path in the code has been executed.
- *Data-flow coverage*. Data-flow coverage checks that every variable definition and usage in the code has been reached.

  There can be distinguished seven data flow criteria: all-defs, all-c-uses, all-c-uses/some-p-uses, all-p-uses, all-p-uses/some-c-uses, all-uses, all-du-path [13].

  Use of the variable is an occurrence of the variable where the variable is referenced.

  Definition-clear path is a sub-path where a variable is not defined.

  All-Uses coverage means that test cases cover definition-clear paths from every definition to every use (c-use and p-use).
- *Edge-pair coverage* means that test cases cover all sub-paths of length two and less in the program.
- *Prime Path coverage* means that test cases cover all prime paths in the program. A prime path is a maximum length simple path. A simple path is a path where no node appears more than once in a path, i.e., it does not have any internal loops.

The disadvantage of these non-mutation code coverage types with respect to fault revelation is that they are *unfalsifiable* as opposed to mutation coverage [35]. In mutation testing, when test suites cannot detect specific mutants, it essentially means that they cannot detect certain types of faults - so test effectiveness is *falsifiable*. For other coverage criteria, test effectiveness can only be falsifiable if we consider it the following way. If tests do not cover a particular item, then the test suite is not effective. However, it is often

not possible to cover everything with tests, and the question of whether there exists any correlation between commonly used testing criteria and fault revelation remains open [15, 35]. Code coverage essentially indicates whether code is covered with any tests rather than how well it is checked.

In contrast to these coverage criteria, mutation testing can be proven to be more effective in fault revelation [35] based on the three main reasons.

First, when we create simple faults in the program using mutation testing and ensure that our test cases can detect those mutants, we ensure that these types of real-world bugs represented by mutants are not present in the program we are testing. This is related to the Competent Programmer Hypothesis that states that developers usually produce the program versions that can be corrected via small syntactic changes.

Second, based on the Coupling Effect hypothesis, we can also assume that our test cases can detect more complex types of faults, which are essentially combinations of simple faults.

Third, when we write test cases that kill mutants, we also test the program under consideration for other types of faults. This is because of the requirements for the mutants to affect observable states of the program. When we create test cases that can kill mutants, we make sure that the faults created by mutant propagation also become observable. This is based on the RIP model for generating test cases [32]. According to the study conducted by Chekam et al. [7], mutant propagation accounts for 36% of the faults revealed by strong mutation testing.

Studies that evaluated the correlation between mutation score and real faults detection are covered in Section 3.2. Studies that compared fault detection capability of mutation testing with other coverage criteria are covered in Section 3.3. The research suggests that mutation testing is more effective in fault detection and, in many cases, subsumes other testing criteria.

2.7 Problems and solutions in mutation testing

While mutation testing is generally considered a powerful fault detecting technique, it has some limitations that prevent it from being extensively used in the industry. The major problems of mutation testing include high computational cost, human oracle problem, and equivalent mutants problem [16]. High computational cost is caused by running a large number of mutants against tests. For m mutants and n test cases, that would require $m*n$ program executions. If there are 100 mutants and one test suite run takes 10 seconds, the total time for all mutants would be 1000 seconds or 16.6667 minutes. In large-scale applications, the numbers can be significantly larger. The second problem of human oracle refers to determining whether the program output is correct in each test case. Although the problem is universal for all test writing processes, it is critical in mutation testing because mutation testing requires writing many test cases. The third problem is explained by the fact that identifying equivalent mutants is an undecidable problem; thus,

additional human effort is required to decide whether a mutant is equivalent and thus possible to kill.

To address these problems, various solutions have been proposed.

To mitigate the problem of high computational cost, a wide range of cost reduction techniques could be used. These cost reduction techniques can be classified into two major categories: reducing the number of mutants and optimizing mutation execution cost.

To reduce the number of mutants, the following techniques could be used: mutant sampling, selective mutation (based on the type), mutant clustering, higher-order mutation, mutant location, and others [16]. Mutant sampling is selecting only a random subset of mutants (e.g., 10%) for mutation analysis. Selective mutation is choosing a minimum required subset of mutant operators that would not significantly affect the test effectiveness. Mutant clustering is selecting a subset of mutants using clustering techniques. Clusters are formed based on the test cases, and only one mutant from each cluster is considered for mutation testing as all the other mutants in the same cluster are assumed to be similar to it. Higher-order mutation is choosing valuable higher-order mutants and working with them only because they subsume first-order mutants from which they were constructed. A mutant location-based technique could be using only one mutant per line of code or identifying patterns of code that contain unproductive mutants [37].

Optimization of the mutation execution cost could be achieved using one of the following options [16]: choosing an optimal killing condition (weak, form, or strong); optimizing run-time (bytecode translation, mutant schema generation, aspect-oriented mutation); using advanced platform support to distribute the computational cost; predicting the results of mutation testing without executing the mutants against the test suite (Predictive Mutation Testing). Predictive Mutation Testing (PMT) [42] uses a machine learning classification model to predict whether a mutant would be killed. PMT has been tested across different programming languages [11] and has different methods proposed over the years that allow achieving AUC value of 0.61 [2].

To resolve the problem of equivalent mutants, various approaches could be used to restrict their number. One of them is Trivial Compiler Optimization (TCE), based on which mutants are considered equivalent only if the compiled code of the mutants is identical to the compiled code of the original version of the program. It is possible to identify approximately 30% of equivalent mutants using this approach [34, 20]. Another method is to use static data-flow analysis in order to reveal parts of the code that can potentially produce equivalent mutants [19].

3 Related work

This section covers related work in mutation testing. First, section 3.1 describes large-scale industrial applications of mutation testing. Next, section 3.2 discusses studies concentrated on assessing the correlation of mutants with real faults. Finally, section 3.3 gives information about the studies that compared mutation testing with other testing criteria in terms of fault detection capability.

3.1 Industrial case studies

Baker and Habli et al. [5] in their study of applying mutation testing in a safety-critical environment, used mutation analysis to identify the most effective mutant types and common causes of failure in the test cases. Based on the analysis, they concluded that mutation testing could point out shortfalls in the test cases that had been peer reviewed.

Ahmed et al. [3] described their experience of applying mutation testing on the Linux kernels RCU, which has around 5,500 lines of code and tests of 1,800 lines of code. They concluded that the technique could show gaps in well-tested modules. As a result of the study, they added three tests and fixed two bugs.

Ramler et al. [41] conducted a case study in a safety-critical system and concluded that strong mutation testing provides valuable guidance towards improving tests in safety-critical systems. They chose a sample of generated mutants in a 60,000-line C program, manually analyzed them, and strengthened the test suites based on the findings.

Petrovic et al. [40] conducted a large-case study of using mutation testing at Google - in an industrial application with more than 30,000 developers and 1.9 million changesets. Mutation testing is integrated into their code review tool and helps authors of code changes and reviewers see potential issues in the code or tests. Mutants are generated only in the changed lines of code. The authors of the paper focused on estimating the costs and benefits of using mutation testing. They found that mutation testing did not add significant overhead to the testing process with their approach and demonstrated that it is impractical and unnecessary to achieve high mutation scores. Instead of using a standard classification of equivalent and redundant mutants, they argue that one could use the classification of productive and unproductive mutants for practical usage (refer to Section 2.5).

In [38], Petrovic et al. explained how they conducted experiments to understand whether mutation testing would have helped prevent bugs. That study is the closest work to this paper in terms of identifying the fault detection capability of mutation testing in industrial applications. They collected the bugs fixed during the last six months and analyzed mutants in the buggy and fixed versions of the program. They concluded that for 70% of the bugs, mutation testing would have found a mutant that survived in the buggy version of the program but was killed in the fixed version of the program, i.e., a fault-coupled mutant. In addition to that, they reported that with mutation testing in place,

developers tend to write more tests and stronger tests (based on mutation score). Another finding was that more than 90% of reported mutants are redundant, which confirmed that their decision to restrict the number of mutants to one per line was a good optimization.

A case study of using mutation testing at Facebook was described by Beller et al. in [6]. In the paper, the authors proposed a way to improve the efficiency of mutation testing by generating only effective mutants learned from the previous errors. In addition to that, they collected feedback from 26 Facebook software developers, 11 of whom stated they would act on shown mutants. The rest presented different reasons not to write a test to kill a mutant, such as the insignificance of the mutant (e.g., logging), the fact that the code was about to be deprecated or improved soon, and insufficient time for writing the test.

3.2 Correlation between mutants and real faults

Most of the studies show a strong correlation between mutants and real faults. However, many studies have limitations: hand-seeded faults, program size, percentage of analyzed mutants, not accounting for code coverage, and others.

Daran and Thevenod-Fosse, in 1996, conducted an experiment studying the correlation between mutants and real faults [8]. It was a safety-critical program with 12 real faults and 24 generated mutants, and the results showed an 85% correlation between mutants and real faults. The limitations of their study are as follows: the experiment was based on one 1000-line C program, 1% of the mutants, and a generated test suite. What is also essential, they did not control for code coverage.

Andrews et al. in 2005 studied the correlation between mutants and real faults as well as artificial faults [4]. They found a strong correlation between mutants and real faults; however, they considered only one program with real faults, which was written in C and had about 6000 lines of code. Furthermore, they used only 10% of the generated mutants for analysis and only automatically generated test cases and did not control for code coverage. Their study was reproduced in 2011 by Namin and Kakarla [28], who used a different mutation testing tool for the same programs. However, Namin and Kakarla found only a weak correlation, and a stronger correlation was observed only with hand-seeded faults.

Just et al. in 2014 conducted a study in which they investigated whether mutants could be a substitute for real faults [17]. They investigated 357 real faults from five open-source projects, and they found that mutants have 73% correlation with real faults. The difference from the previous studies is that they used large programs, real faults, developer-written and automatically-generated test suites, and controlled for code coverage. Their approach was as follows: they isolated all bugs, performed mutation analysis on the buggy and fixed versions of the program, and analyzed whether mutation score increased in the fixed version.

Papadakis et al. [36] have found only a weak correlation between mutation score and fault detection when controlling for test suite size. However, a combination of test suite size and mutation score provided good results.

3.3 Correlation between different test coverage criteria and fault detection

Most of the studies that compared mutation testing with other test coverage criteria concluded that mutation testing is the most effective in detecting real faults and often subsumes other test coverage criteria. The descriptions of the coverage criteria mentioned in the studies can be found in Section 2.6.

Offut et al. [31] and Frank et al. [12] both compared fault detection rates of two testing criteria - all-uses data-flow and mutation coverage - and concluded that mutation-adequate test sets reveal more faults than all-uses. Li et al. [25] found that mutation testing reveals more faults than edge-pair, all-uses, and prime path testing criteria (85% as opposed to 65%). They used seeded faults and manually generated tests and compared the effectiveness (ability to detect faults) and efficiency (number of tests required to reveal faults) of the mentioned testing criteria. They concluded that mutation performed better with regard to both effectiveness and efficiency. However, Kakarla et al.[18] performed a meta-analysis of the previous studies and found that mutation-based testing is two times more effective than data-flow testing but three times less efficient. Baker and Habli et al. [5] conducted an empirical study of mutation analysis in safety-critical software written in C and Ada, which achieved high statement, branch, and modified condition coverage. They concluded that mutation testing identified shortfalls not detected by the traditional coverage criteria and code review. Just et al. [17] also found that the mutants have a higher correlation with real faults than statement coverage. Chekam et al. [7] conducted an empirical study in which they evaluated the fault revelation power of statement, branch, weak mutation, and strong mutation. They found that a strong mutation has a stronger correlation with fault revelation than other code coverage types, which have only weak or no correlation with fault revelation. However, this fault revelation capability of strong mutation testing comes into power only after a certain threshold of mutation coverage is achieved.

Ahmed et al. [3] used a large number of open-source Java programs to study the correlation between statement coverage and mutation score with bug fixes and found only a weak negative correlation for both.

4 Methodology

This section covers the methodology applied in the study. Section 4.1 describes how mutation testing would be used in the systems if proven to be helpful. Section 4.2 elaborates on the reasons for the choice of mutant set generation. Section 4.3 depicts the procedures for the experiments with the past bugs based on the defined potential application plan. Section 4.4 briefly describes subject services and Section 4.5 gives information about the mutation testing tool.

4.1 Mutation testing application plan

In order to conduct experiments to see whether mutation testing would have helped detect past software bugs, the experiments have been conducted in a way mutation testing would have been applied if introduced in the systems.

If integrated into the systems, mutation testing would be applied to the changed files in every pull request introducing a change. Since the mutation score is distorted by equivalent and redundant mutants, there would be no requirements for the minimum mutation score. Instead, the focus would be only on the survived mutants.

The testing would be applied before the code review because the primary goal is for developers to take action based on the mutation testing report before the changes get merged and deployed to production. In addition to that, the report helps reviewers to be more confident in the reviewed code.

After creating a pull request, the developer can choose to run mutation testing (e.g., using a specific command) and see the created mutation testing report on a separate page. Based on the mutation testing report for the pull request, the developer would try to kill survived mutants by augmenting or changing the existing test suite. In some cases, this would not be possible because the mutants would be equivalent.

As a result of the mutation testing, the developer would be able to

- strengthen the test suite by adding new test cases or changing existing test cases to kill mutants and thus prevent future bugs
- locate current bugs during the process of mutation testing

4.2 Justification for the choice of the mutant set

The proposed approach applies mutation testing on the files that contain changes in pull requests rather than the whole system. While applying mutation testing on the entire set of mutants in the system would give confidence that all mutants related to the change are killed, it is undesirable and impractical for a few reasons.

First, it is not efficient in modern software development with continuous integration and continuous deployment practices. As there can be several deployments per day in one

service, one run of mutation testing would be computationally expensive and require a fair amount of developer time for considering each mutant, adding tests, and verifying the correctness of the result. Second, a developer would want to focus only on the mutants relevant to their code change. This is important, as the testing should be constructed to be convenient for the developers to integrate the tool's usage in their development flow. Third, by adding test cases to kill mutants in the code parts not relevant to the change, it is possible to increase the mutation score but fail to address the most critical mutants directly related to the code change.

Therefore, in this use case, the plan is to apply mutation testing to changed files in a pull request. Petrovic et al. [40] and Beller et al. [6] further restrict the area of mutation testing application by mutating only diffs in pull requests, rather than the whole files. While this allows reducing developer work further and minimizes computational cost, it also creates a possibility to skip some critical mutants relevant to the change but not included in the changelist. For example, suppose a developer changed method A in their pull request. This method A is called by another method B. By focusing only on commit changes, the developer would only see the mutants generated inside method A. However, it could be that killing mutants inside method B would help find a bug inside method A. The question now arises whether method B is necessarily in the same file as method A. Most interdependent methods in subject services are contained in one file. However, some of the mutants relevant to the change may be located outside the changed files and thus outside the proposed application of mutation testing. Nevertheless, this trade-off between usability and cost-effectiveness is necessary for the process to be smoothly integrated into the industry.

In a recent study by Ma et al. [26], an algorithm for determining commit-aware mutants was developed. Commit-aware mutants are mutants that are relevant to the code changes. Using test execution outputs and mutant matrices containing information about the mutants killed by test cases, the authors compute the correlation between the mutants and the code change using a specific algorithm and create a set of commit-aware mutants collected from the whole project. However, since this algorithm is not available in the mutation testing tool and would require a considerable amount of work to implement and validate, the potential use of mutation testing is defined based only on currently available tools.

4.3 Experiments with past bugs

The experiments with the past software bugs were conducted to find whether mutation testing would be beneficial for our systems. In particular, the aim was to see whether mutation testing could have helped prevent the faults if introduced in the systems before the bugs got into production.

There are two ways mutation testing could be helpful in bug prevention. First, as mutation testing helps to strengthen test suites, the first way is to apply mutation testing

on the version of the program before the bug was introduced and add the tests to kill survived mutants if possible. The mutation testing could have been proven to be helpful if these newly added tests would fail on the version of the program with the bug. The second way is to apply mutation testing on the bug-introducing version of the program, as during the process of writing tests to kill the mutants, it is also possible to locate a fault. Thus, a program version before the bug introduction and a bug-introducing program version were identified for each bug. Afterwards, the tests to kill the survived non-equivalent mutants were added to see if any of them would fail. If at least one test failed, it was concluded that mutation testing would have prevented the bug. Otherwise, mutation testing was considered not helpful for that particular bug.

4.3.1 Locating past bugs

A bug tracking system Jira was used to locate past bugs. Bugs in Jira are specified under a particular ticket type called "Bug", which has a "Description" section where the system, the faulty behavior, and the desired behavior are described. The types of such bugs can be different: bugs resulting from faulty business requirements (missing requirements, miscommunication, incorrectly stated requirements), logical design errors, or development errors. A corresponding pull request that fixes the described bug can be found using the version control system Git for each of these tickets.

The following procedure was followed to locate the set of bugs for experiments:

1. Identify all bug tickets for the subject systems.
2. Remove bug tickets that do not have associated code (e.g., bugs that had been resolved during the implementation of other tasks, that were not reproduced, duplicate bugs, bugs that were resolved by a fix in another system).
3. Remove bug tickets without any description and involving major code changes.

For the resulting set, bugs corresponding to the bug tickets were identified and prepared as described in the next section.

4.3.2 Preparing past bugs for experiments

For each bug, the following steps were conducted to prepare the bug for experiments.

First, two pull requests - a bug-introducing pull request and a pre-bug-introducing pull request - were identified to represent exactly how the bug could have been caught if mutation testing was introduced in the systems on the pull request level. A *bug-introducing pull request* is a pull request that introduced the bug in a system. We are interested in a set of files associated with the bug that were changed in that pull request. A program version that corresponds to the bug-introducing pull request is a *bug-introducing*, or *buggy*, program version. A *pre-bug-introducing pull request* is a pull request that last

changed bug-related file(s) before the bug was introduced in them. A program version that corresponds to the pre-bug-introducing pull request is a *bug-free version* before the bug introduction. The relations between the program versions are illustrated in Figure2.

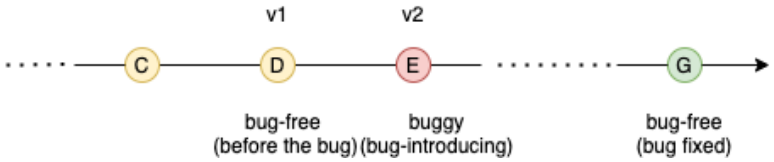

Figure 2. Program versions. Option 1.

In many cases, the bug was introduced in the first version of the associated files, i.e., the initial implementation of the functionality contained an error. In that case, a pre-bug-introducing pull request for those files and the corresponding bug-free program version do not exist. Thus, there is only one, *bug-introducing version*. This option is illustrated in Figure3.

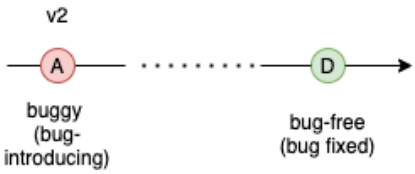

Figure 3. Program versions. Option 2.

Before running the mutation testing tool, all compiler issues for the programs were resolved. The program versions were up to three years old; therefore, many libraries, frameworks, and services referenced in the project got updated, which in many cases caused errors when compiling the projects and running tests. After all compile errors were resolved, the tests were run to verify they are passing. It is known that the tests were passing at the time of the commits because passing tests are checked automatically in the deployment pipeline. Therefore, the tests may only fail during experiments because of the outdated versions of the projects. It is also impossible to apply the mutation testing tool to the program if all tests do not pass. Thus, having passing tests is also a prerequisite for running a mutation testing tool.

4.3.3 Experiment procedures

In this section, two procedures for the experiments are described. The first procedure was applied when there were two versions - a bug-free version v1 and a buggy version v2. The second procedure was followed when there was only a buggy program v2.

For each bug, there exists a buggy version of the program v2 and a corresponding test suite t2. All tests in t2 for v2 pass; otherwise, the bug in v2 would have been found.

For some bugs, there exists a bug-free version of the program v1 and a corresponding test suite t1. All tests in t1 for v1 also pass.

In cases when there were working versions v1 and v2 with the respective test suites t1 and t2, the procedure shown in Figure 4 was followed.

First procedure.

1. Run the mutation testing tool on v1 using t1 and get a set of mutants M1, where m1 is a subset of survived non-equivalent mutants.

2. If all non-equivalent mutants are killed, i.e., m1 is empty, skip steps 3-5 and go to step 6. In this case, t1 seems to be strong enough, although it is known it did not prevent the bug that appeared later in v2. Still, there is a possibility that applying mutation testing on v2 using t2 (t2' = t2) can help.

3. Add a set of tests nt1 to the original test suite t1 and get t1'. nt1 consists of tests that kill survived non-equivalent mutants from m1. So, `t1' = t1 + nt1`.

4. Merge nt1 with t2. Compare the set of mutants that survived in v1 using t1 (m1) with the set of mutants that survived in v2 using t2 (m2). The set of mutants m2 may consist of m1 and some additional mutants, but it can also contain an incomplete set of m1 since the code could have changed such that some of the old mutants disappeared. Add those newly added tests from nt1 to t2 for which the mutants exist in v2. This modified version of t2 would be called t2'.

5. Run t2' on v2 and check if it passes (v2(t2') correct). If t2' fails on v2, then we can conclude mutation testing helped detect the bug. Otherwise, continue the process.

6. Run the mutation testing tool on v2 using t2' and get a set of mutants M2, where m2 is a subset of survived non-equivalent mutants.

7. If there does not exist a set of survived non-equivalent mutants m2 in M2, then we can conclude that mutation testing would not have helped detect the bug since, based on its results, the test suite would seem strong enough at this point, and the bug indicates that it is not.

8. Add additional tests nt2 to t2' to kill mutants from the set m2 and get t2". The mutants to kill are the mutants that were not in version v1 but appeared in v2; thus, tests for them are absent from t2' at this point.

9. Run t2" on v2 and check if it passes (v2(t2") correct). If all tests pass, then mutation testing did not help to detect the bug. If at least one of the tests fails, mutation testing helped detect the bug.

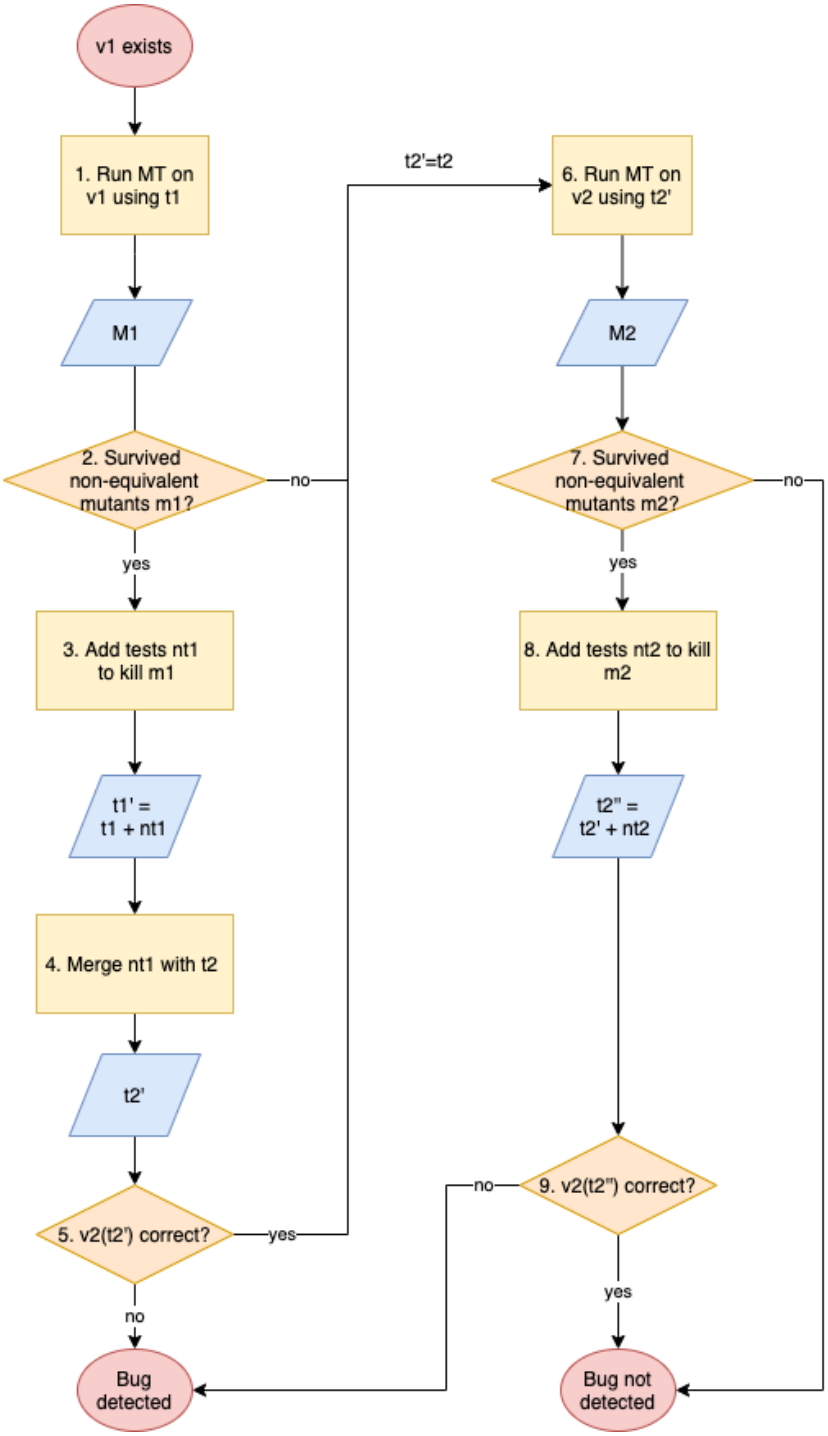

Figure 4. Experiment procedure 1

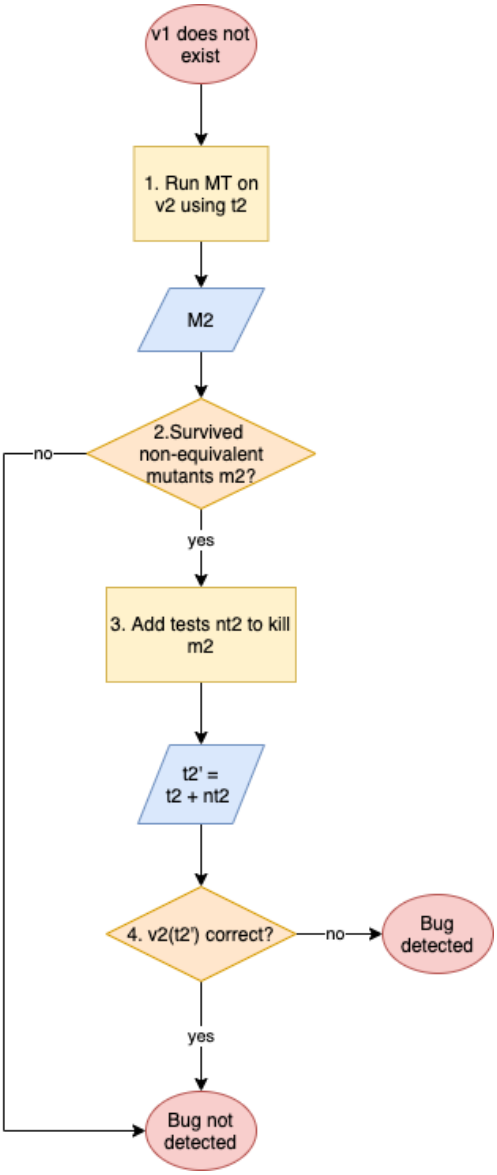

Figure 5. Experiment procedure 2

In cases when there was only version 2 with a respective test suite t2, the procedure shown in Figure 5 was followed.

Second procedure.

1. Run the mutation testing tool on v2 using t2 and get a set of mutants M2, where m2 is a subset of survived non-equivalent mutants.

2. If there does not exist a set of survived non-equivalent mutants m2 in M2, then

we can conclude that mutation testing would not have helped since, based on its results, the test suite t2 would seem strong enough, and the bug indicates that it is not.

3. Add a set of tests nt2 to the original test suite t2 and get t2'. nt2 is a set of tests that kills survived non-equivalent mutants from m2. So, `t2' = t2 + nt2`.

4. Run t2' on v2 and check if it passes (v2(t2') correct). If all tests pass, then mutation testing did not help to detect the bug. If at least one of the tests fails, mutation testing helped detect the bug.

Running the mutation testing tool included the following steps:

1. Adding stryker4s sbt plugin

2. Specifying configuration: files to mutate and test suites to run

3. Running the tool

4.4 Subject programs

The focus of the study is three industry services written in Scala. General information about their sizes and the number of tests in each of them is given in Table 1.

Table 1. Subject programs

Service	Service size (LoC)	Number of tests
Ratedeck Parser	5640	255
Cost API	7722	289
Provider Service	10571	497

Each service has a version control system and integration with the bug tracking system, allowing locating the bugs and performing mutation analysis.

There is a multi-layer test suite for each given system and a code review for each change in a system. The types of tests included in the multi-layer test suite are unit, integration, load, chaos, and end-to-end tests. In addition, code review is required for any change that is made in a system.

It is possible to run a coverage calculation for each system, which would return statement and branch coverage scores.

Any change in the system can be deployed to production only if all tests in the given system pass. Thus, all the bugs that are considered in this study have been deployed with passing tests.

4.5 Mutation testing tool

The testing framework that was used is stryker4s[2], a Scala framework for mutation testing [1]. It is the only active mutation testing framework for Scala. The framework has a wide range of supported mutant operators and metrics.

From the implementation point of view, it does the following:

1. Create the mutants and compile the code once.
2. Collect information about the tests from sbt.
3. Start a separate worker-process for each mutant and sent the test setup information to it.
4. For each worker-process, instantiate a test framework and run the tests.

To compile code only once, stryker4s uses a Mutation Switching technique. It works the following way: First, all mutants are identified for the whole program. Second, identified mutants are introduced into the codebase using Pattern Matching in Scala. Third, introduced mutants are tested one by one in isolation by changing the environmental variable associated with each mutant. Here is an example of how Mutation Switching works:

This is the original program that we want to mutate.

```
def canGetPension(p : Person) : Boolean = {
    p.age >= 63
}
```

Mutants are introduced into the source code in the way shown in the following piece of code. The default case corresponds to the original version of the program, and all the other cases are mutants.

```
def canGetPension(p : Person) : Boolean = {
  stryker4s.activeMutation match {
    case Some("0") =>
        p.age > 63
    case Some("1") =>
        p.age < 63
    case Some("2") =>
        p.age == 63
    case _ =>
        p.age >= 63
}}
```

[2]https://github.com/stryker-mutator/stryker4s

4.5.1 Configuration

The setup of the tool is straightforward. It could be added as an sbt or maven plugin in the project. By default, it will mutate all files and run all tests against each mutation. However, it is also possible to add a configuration file. General configurations used during the experiments are the following:

- **mutate** - list of files to mutate
- **test-filter** - list of tests to use for mutation testing
- **reporters** - reporters to use (console, html, json, and dashboard)
- **base-dir** - the directory from which to start; should be used in multi-module projects
- **excluded-mutations** - specify the mutant operators to exclude from the list; was used when a certain operator caused the tool failure

4.5.2 Mutant operators

The following mutant operators (mutators) are available in the framework:

- Boolean Literal

Original	Mutated
true	false
false	true

- Conditional Expression

Original	Mutated
while (a>b)	while(false)
do while (a > b)	do while (false)
if (a > b)	if (true)
if (a > b)	if (false)

- Equality Operator

Original	Mutated	Original	Mutated
a < b	a <= b	a > b	a >= b
a < b	a >= b	a > b	a <= b
a <= b	a < b	a >= b	a > b
a <= b	a > b	a >= b	a < b
a == b	a != b	a != b	a == b

- Logical Operator

Original	Mutated
a && b	a || b
a || b	a && b

- Method Expression

Original	Mutated	Original	Mutated
a.filter(b)	a.filterNot(b)	a.filterNot(b)	a.filter(b)
a.exists(b)	a.forall(b)	a.forall(b)	a.exists(b)
a.take(b)	a.drop(b)	a.drop(b)	a.take(b)
a.takeRight(b)	a.dropRight(b)	a.dropRight(b)	a.takeRight(b)
a.takeWhile(b)	a.dropWhile(b)	a.dropWhile(b)	a.takeWhile(b)
a.isEmpty	a.nonEmpty	a.nonEmpty	a.isEmpty
a.indexOf	a.lastIndexOf(b)	a.lastIndexOf	a.indexOf(b)
a.max	a.min	a.min	a.max
a.maxBy(b)	a.minBy(b)	a.minBy(b)	a.maxBy(b)

- String Literal

Original	Mutated
"foo" (non-empty string)	"" (empty string)
"" (empty string)	"Stryker was here!" (non-empty string)

4.5.3 Mutants and Metrics

The following mutants can be the result of mutation testing:

- **Killed** - a mutant that caused a test failure
- **Survived** - a mutant that did not cause any test failures
- **No coverage** - a mutant located in a piece of code not covered by any tests
- **Timeout** - a mutant caused the tests to run too long; e.g. created an infinite loop
- **Runtime error** - a mutant caused a runtime error
- **Compile error** - a mutant caused a compile error
- **Ignored** - a mutant that was skipped due to configuration or another reason; e.g. could not be tested

Calculated metrics are shown in Table 2.

Table 2. Metrics calculated in stryker4s

Metric name	**Formula**	**Description**
Detected	killed + timeout	mutants detected by tests
Undetected	survived + no coverage	mutants not detected by tests
Covered	detected + survived	mutants that have statement coverage
Valid	detected + undetected	not ignored mutants that did not result in an error
Invalid	runtime errors + compile errors	mutants that resulted in an error
Total mutants	valid + invalid + ignored	all mutants
Mutation score	detected / valid * 100	percentage of killed mutants out of all valid mutants
Mutation score (covered)	detected / covered * 100	percentage of killed mutants out of all covered mutants

5 Results

The results of the experiments for each of the services - Ratedeck Parser, Cost API, and Provider Service - are described in this section. For each service, the tables with the bug descriptions and results of mutation testing are presented. Each table has the following columns:

- **ID** - an assigned bug identifier
- **Description** - a brief description of the bug
- **Covered** - a column indicating whether the buggy code was covered with tests based on the statement and branch coverage criteria. These criteria are combined because the results for both were the same. This is explained by the fact that in the Scala systems, conditional statements are rarely used, and if the bug was covered, it was covered based on both criteria.
- **MT results v1** - mutation testing results for the bug-free version v1
- **MT results v2** - mutation testing results for the buggy version v2
- **Bug detected** - whether mutation testing could help detect the bug based on the procedure described in Section 4.3.3.

For MT results v1 and MT results v2, numbers in brackets mean the following: [**Valid mutants**, **Detected mutants**, **Mutation score**], where Mutation score is Detected mutants / Valid Mutants (see Section 4.5.3). In these experiments, only in two cases, the number of total mutants was not equal to the number of valid mutants. In both cases, a mutant was ignored because a static mutant could not be tested:

```
private val tableName: String = "sms_rates"
```

Other possible values in the table include "n/a", "compile error", and "NaN% for mutation score.

- **n/a** - not applicable, present in column "MT results v1" when there was no v1 for a bug and the second procedure was followed.
- **compile error** - no results because there were issues with compiling the outdated project version. In that case, the result for the last column, "Bug detected", was based on the manual analysis of the bugs and the test suites.
- **NaN%** - mutation score is not available because no mutants were generated.

During the experiments, all survived mutants have been evaluated, and 37 unique tests were written. The tests are contained in private repositories. The examples of written tests are presented in Section 5.1, and examples of analyzed mutants are given in Section 6.

5.1 Service 1: Ratedeck Parser

Ratedeck Parser is a service that reads and parses carriers' price lists called ratedecks, makes computations of different types of rates using information from other services, and stores the results in the database. The information about the service size, number of tests, and statement and branch coverage is shown in Table 3.

Table 3. General information about Ratedeck Parser

Service	LoC	Number of tests	Statement coverage	Branch coverage
Ratedeck Parser	5640	255	66.27%	65.18%

Based on the criteria described in Section 4.3.1, 10 bugs have been identified associated with 12 bug fixes. For one bug, which is a faulty algorithm, three fixes were made. The results obtained after applying mutation testing using the approach described in Section 4.3.3 are shown in Table 4.

From the table, it can be seen that only two bugs out of 10 had v1, which is the version of the associated files before the bug introduction. This means that in most cases, the bug was introduced when creating the file(s) with the given feature. In other words, the initially developed feature contained a flaw. Regarding statement and branch coverage, out of 10 bugs, 3 have not been covered by any tests.

For 2 bugs - 7 and 10 - it was impossible to compile the project and set up the tool as they were contained in the very old versions of the code. Instead, manual analysis has been conducted to check for possible mutants, and the results for these bugs are presented based on that.

Bug detection. The results indicate that for 9 out of 10 bugs, mutation testing did not help to find a bug. This means that newly added tests that killed all survived non-equivalent mutants did not detect the bugs.

For 1 bug out of 10 (bug 3), mutation testing helped to find one problem with a faulty algorithm. This algorithm has been entirely covered based on the statement and branch coverage criteria. The results are presented for the initial bug-introducing version and two versions after the first and second fixes, which are slightly modified but still incorrect versions of the program. For the last two versions, mutation testing generated the same fault-coupled mutant. Thus, mutation testing would have helped to identify an issue with the algorithm earlier and minimize the number of attempts to fix the program.

Let us examine bug 3, where mutation testing has proven to be helpful. The method that contained survived mutants is shown in Figure 6 in the form of the mutation testing report.

From the bug fix, it is known that the wrong part of the code was in another method (*computeCountryFallback*), but that method did not contain any survived mutants. At the

Table 4. Results of experiments for Ratedeck Parser

ID	Description	Covered	MT results v1	MT results v2	Bug detected
1	Wrong method return type	Yes	[79,56,70.89%]	[93,71,76.34%]	No
2	Empty vector passed instead of failing	No	n/a	[42,31,73.81%]	No
3	Incorrect algorithm for calculating specific rates	Yes	n/a	[13,10,76.92%], [18,13,72.22%], [21,17,88.95%]	No Yes Yes
4	Incorrect algorithm for calculating specific rates with different dates	Yes	n/a	[17,12,70.59%]	No
5	Integration problem with another service	Yes	n/a	[14,13,92.86%]	No
6	Incorrect processing of files (wrong assumption about the contents)	Yes	n/a	[17,11,64.71%]	No
7	Incorrect log level for an error	No	n/a	compile error	No
8	Certain country codes absent from a library	Yes	n/a	[1,0,0%]	No
9	Certain country codes absent from a library	Yes	n/a	[1,0,0%]	No
10	Integration problem with another service	No	compile error	compile error	No

same time, five mutants (2, 3, 4, 7, 9) survived in the method *fallbacks*, which eventually called the buggy method *computeCountryFallback*.

The survived mutants are the following:

- Mutant 2 is Equality Operator: "==" –> "!="

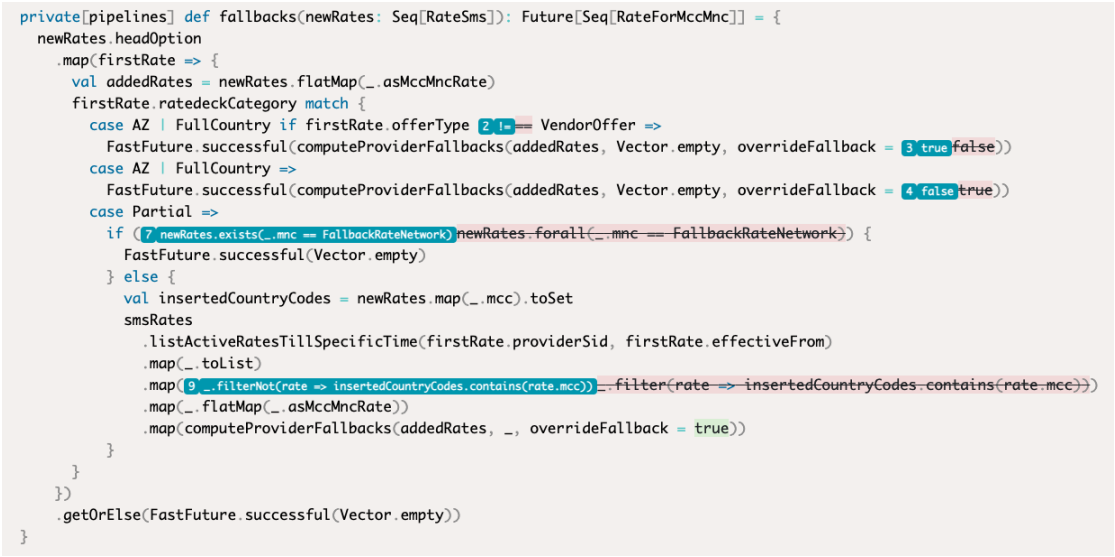

Figure 6. Mutation Testing Report for Bug 3, method fallbacks

- Mutants 3 and 4 are Boolean Literal operators: "false" –> "true" and "true" -> "false"
- Mutants 7 and 9 are Method Expression operators: "forall" –> "exists" and "filter" –> "filterNot"

By following the procedure described in Section 4.3, the following results were obtained:

- Two existing test cases were fixed such that they killed mutants 2, 3, and 4. The first test case killed Mutant 2 and Mutant 3 (joint mutants), and the second test case killed Mutant 4. These test cases existed before and were supposed to kill those mutants based on their descriptions. However, because of the inaccurately chosen test data, it was revealed that the test cases were not verifying the stated behavior and therefore were misleading and ineffective. The original and the fixed versions of the test cases can be found in Appendix in Figures 13 - 16.
- One test case was added to kill Mutant 9 (Figure 17 in Appendix).
- Another test case was added to kill Mutant 7 (Figure 18 in Appendix). This test case failed when running on the buggy version of the program. The failure indicates this was a fault-coupled mutant that would have helped identify an algorithm issue.

The resulting mutation testing report after fixing and adding test cases is shown in Figure 19 in Appendix. The report shows that all the mutants except for the fault-coupled Mutant 7 were killed; it was impossible to rerun the mutation testing tool with the newly added test for Mutant 7, as it failed on the original program.

5.2 Service 2: Cost API

Table 5. General information about Cost API

Service	LoC	Number of tests	Statement coverage	Branch coverage
Cost Api	7722	289	83.75%	76.47%

Cost API is an API service that makes the rate data available to internal stakeholders through various endpoints. The information about the service size, number of tests, and statement and branch coverage is shown in Table 5.

Based on the criteria described in Section 4.3.1, 7 bugs have been identified for this service. The results of the experiments described in Section 4.3.3 are shown in Table 6.

Table 6. Results of experiments for Cost API

ID	Description	Covered	MT results v1	MT results v2	Bug detected
1	Missing optional argument	Yes	n/a	[12,12,100%]	No
2	Integration problem with another service	Yes	[6,5,83.33%]	[9,8,88.89%]	No
3	Incorrect endpoint return value	Yes	n/a	[30,27,90%]	No
4	Incorrect business logic implementation	Yes	n/a	[24,18,75%]	No
5	Integration problem with front-end	No	[23,16,69.57%]	[23,16,69.57%]	No
6	Invalid input not rejected	Yes	[0,0,NaN%]	[10,10,100%]	No
7	Calculation in seconds instead of minutes	Yes	n/a	[24,18,75%]	No

The table shows that 3 bugs had a program version before the bug, and 4 bugs had only the buggy version. Only 1 bug out of 7 was not covered with any tests based on the statement and branch coverage criteria.

For one bug (bug 4), mutation testing helped to reveal inefficiency in code (more on this in the refactoring example in Section 6.1).

Bug detection. The results indicate that mutation testing did not help detect any of the considered bugs. In other words, after adding the test cases to kill all survived non-equivalent mutants that were possible to kill, none of the bugs were detected. For this service, many mutants were mutating database table column types (e.g., "DATETIME") in Data Access Object classes; the tests for those mutants were not added.

5.3 Service 3: Provider Service

Table 7. General information about Provider Service

Service	LoC	Number of tests	Statement coverage	Branch coverage
Provider Service	10571	497	93.38%	78.00%

Provider Service is an API service that makes the providers' data available to internal stakeholders through various endpoints.

For Provider Service, no code-related bugs have been identified based on the defined criteria described in Section 4.3.1. The only found bugs were related to library updates and performance issues. The lack of the bugs could be explained by the fact that provider Service is an API service developed a long time ago.

Based on that, it was not possible to conduct experiments with the past bugs. However, mutation testing has been run on the whole service instead. Out of 736 valid mutants, 594 mutants have been detected by the tests with a mutation score of 80.71%. This service is the most covered based on statement and branch coverage percentages, and a relatively high mutation score proves that the service is well-tested.

6 Analysis of mutants

During the process of mutation testing, a developer needs to analyze each mutant and decide whether or not the test could and should be added based on the types of the mutants.

In this section, analysis of mutants is presented based on the more general classification of equivalent and redundant mutants described by Papadakis et al. [35] as well as a more practical classification of productive and unproductive mutants introduced by Petrovic et al. [40]. All these types of mutants are discussed in detail in Section 2.5.

6.1 Equivalent and redundant mutants

Many of the survived mutants were equivalent mutants. By default, equivalent mutants are unproductive since they waste developer time to be analyzed but cannot have corresponding tests. However, some of them can reveal code redundancy leading to code refactoring, and thus, be productive.

An example of a piece of code with two equivalent mutants is shown in Figure 7.

```
val tierVolumeDifferences = (0L +: planVolumes.filter(_ <= currentTierVolume))
  .sliding(2)
  .map {
    case Seq(prevTier, nextTier, _*) =>
      if (nextTier 49 >= > eomVolume || (nextTier == planVolumes.max && nextTier 56 <= < eomVolume))
        eomVolume - prevTier
      else
        nextTier - prevTier
  }
  .toVector
```

Figure 7. Equivalent mutant example. Ratedeck Parser.

These two mutants are of type *Equality Operator*: Mutant 49 changes ">" to ">=" and Mutant 56 changes "<" to "<=". When `nextTier` is equal to `eomVolume`, the original version of the program goes to the `else` branch and returns `nextTier - prevTier` value. If we modify the program such that we replace ">" with" >=" or "<" with "<=", then the program goes to `if` branch and returns `eomVolume - prevTier`. This is equivalent to `nextTier - prevTier` as `nextTier = eomVolume`. Thus, these are equivalent mutants.

Another equivalent mutant example is shown in Figure 8. The mutant is of type *Method Expression* and changes `exists` to `forAll`. In this particular case both methods produce the same result because `p.effectiveTo` is of type `Option[OffsetDateTime]`, so `p.effectiveTo.exists` and `p.effectiveTo.forall` are semantically equivalent and refer to `OffsetDateTime`.

Productive equivalent mutants. Let us consider a bug where mutation testing and particularly equivalent mutants helped to notice inefficiency in code. The example is shown in Figure 9.

```
def filterPlans(
  plans: Vector[TieredPlanV2],
  datetime: OffsetDateTime = OffsetDateTime.now()
): Vector[TieredPlanV2] = {
  // TODO: should be static, with proper name
  plans
  // effectiveFrom <= now()
    .filterNot(_.effectiveFrom.isAfter(datetime))
    // effectiveTo is null || effectiveTo >= now()
    .filter(p => p.effectiveTo.isEmpty || 66 p.effectiveTo.forall(!_.isBefore(datetime))p.effectiveTo.exists(!_.isBefore(datetime)))
}
```

Figure 8. Equivalent mutant example. Ratedeck Parser.

```
callDuration match {
  case x if x 0 <= 0                                          => 0
  case x if x 3 >= > 0 && x < minDuration                      => 0
  case x if x >= minDuration && x 14 <= initDuration           => initDuration
  case x if x 17 >= > initDuration && x 21 <= additionalRounding => additionalRounding
  case _                                                      => callDuration
}
```

Figure 9. Refactoring example. Cost API. Original version

All survived mutants (0, 3, 14, 17, 21) are equivalent mutants of type *Equality Operator*. However, the first two - Mutant 0 and Mutant 3 - can point out possible improvement in code, while others are ordinary equivalent mutants.

Mutant 0 is replacing "<=" with "<" when comparing to 0. In the original program, when x is equal to 0, it matches the first case and returns 0. If we replace "<=" with "<" and try to match case when x = 0, it would match the default case, which is `callDuration`, 0. Mutant 3 is replacing ">" with ">=" when comparing to 0. In the original program, when x is equal to 0, it matches the first case and returns 0. With this mutant, it would still return 0.

It could be noticed that the first two cases could be combined as shown in Figure 10.

```
callDuration match {
  case x if x < minDuration                          => 0
  case x if x >= minDuration && x 7 <= initDuration  => initDuration
  case x if x 10 >= > initDuration && x 14 <= additionalRounding => additionalRounding
  case _                                             => callDuration
}
```

Figure 10. Refactoring example. Cost API. Refactored version

As for redundant mutants, a few of them have been noticed among the survived mutants. However, redundant mutants can be present among the killed mutants, too; those were not subject to analysis. Redundant mutants are generally neither productive nor unproductive since they get killed with other mutants. Their major disadvantage is the influence on the mutation score. However, if a developer spends some time on a

particular mutant A that is harder to analyze than its joint mutant B, then examining the first mutant first would lead to unproductively spent time. It would be more beneficial to examine the second mutant first. Thus, the notion of productivity of redundant mutants is mainly dependent on the development process.

An example of a redundant mutant could be taken from Figure 6. As described in section 5.1, two mutants (Mutant 2 and Mutant 3) in `fallbacks` method were killed with one test, so we could say that Mutant 3 was redundant.

In many cases, redundant mutants were the mutants that mutated the same piece of code. For example, mutants of type Equality operator (">" -> ">=" and ">" -> "==") can be usually killed with one test.

6.2 Non-equivalent and non-redundant mutants

Most non-equivalent mutants are productive, as it is possible to write effective test cases for them.

One such example is shown in Figure 11. In this case, Mutant 26 survived, which mutated `planVolumes.min` to `planVolumes.max`. Therefore, a test case was added during the process of mutation testing where the value of `volume` was greater than in `planVolumes`.

Other examples are also shown in Figure 6. Mutants 2, 4, 7, and 9 had corresponding tests, and one of them helped us to find an existing bug, thus proving its usefulness.

```
val currentVolume = planVolumes.find(_ >= volume).getOrElse(26 planVolumes.min planVolumes.max)
```

Figure 11. Non-equivalent mutant example. Ratedeck Parser.

However, some unproductive non-equivalent and non-redundant mutants can exist, for which developers usually would not want to write test cases.

Unproductive non-equivalent and non-redundant mutants. Most unproductive mutants resulted from String Literal mutations, which is replacing strings with empty strings.

An example of a mutant that could arguably be considered unproductive is a mutant that replaces a log statement with an empty string. There were many such mutants, and one example is shown in Figure 12. Addressing the mutant would result in a hard-to-maintain test that will take developer time while not bringing much value.

```
val volume = currentVolume(smsNetworkVolumes)
logInfo(29 "" s"[TieredPlans Scheduler] current volume is: $volume for plan wiht id: ${plan.id}.")
```

Figure 12. Unproductive mutant example. Ratedeck Parser.

Other examples of unproductive mutants discovered during the experiments were mutating a URL string and database table column types. Most of the survived mutants in Cost API files were of this type.

7 Discussion

7.1 Answers to the research questions

The results have shown that the main research question **RQ** (*Does mutation testing improve the fault detection capability of the test suites used by Twilio*?) does not have a simple yes or no answer. Since there was at least one case where mutation testing was helpful, mutation testing improves the fault detection capability of the test suites. However, since the percentage of such cases where mutation testing was beneficial is small, it could be argued that the improvement based on it is insignificant.

For **RQ1** (*If the answer to RQ is "no", what are the reasons why mutation testing does not improve the fault detection capability of the considered test suites?*), two main reasons could be provided explaining why mutation testing was not helpful for most bugs.

First, some bugs could not be caught at the unit testing level, such as integration problems with other services or library limitations. In the first service, out of 11 bugs for which mutation testing was not helpful, 2 were associated with integration with other services, and 2 - with the limitations of third-party services. In the second service, out of 7 bugs for which mutation testing was not helpful, 2 were associated with the integration with other services. In addition to that, one bug resulted from the misinterpretation of business requirements, and some bugs may have resulted from missing business requirements. It is not possible to state precisely when missing requirements were the sources of the bugs because quite often, not all business requirements are listed in Jira tickets. Many requirements are implicit; they are based on the shared knowledge and can be found in the general documentation and discussed within a team.

Second, the level of complexity of the bugs could also be a factor. For the first service, Ratedeck Parser, with more bugs in total, it was proven to be helpful in one case, while for the second service, Cost API, mutation testing was not helpful at all. The difference between the two is that Ratedeck Parser contains more complex logic than API services, such as Cost API and Provider Service. Based on that, it could also be suggested that mutants can be helpful in cases where complex logic that requires extensive testing is involved.

For **RQ2** (*If the answer to RQ is "yes", which mutant operators are the most effective in detecting faults?*), it is not possible to draw a conclusion as there was only one fault-coupled mutant.

7.2 Identified benefits and barriers

The results of the experiments suggest that mutation testing can bring small value in terms of finding bugs. Although the study was focused on identifying the fault revelation power of mutation testing, it was discovered throughout the process that mutation testing could bring other benefits, such as finding inefficient test cases and improving code

quality. Moreover, based on the author's opinion, many tests, although not associated with past bugs, would still be beneficial to have in the test suites to prevent future bugs.

Nevertheless, there exist several substantial impediments to using mutation testing at the moment.

1. First, immaturity of the tool. For mutation testing, a programming language and a mutation testing tool matter. The mutation testing tool for Scala is still in the development phase and is not ready to be used in big software projects. For example, it fails when the program contains XML code; in one of the services, there is a considerable amount of XML code. Another example is that the tool fails when there are specific structures. For instance, the mutation testing tool failed when it tried to change "<=" to "==" because the correct mutant was "===" in that case. Moreover, it fails to mutate some pieces of code displaying warning logs about that. Nevertheless, it should be mentioned that the tool is actively developing. During the experiments, a bug with configuration (test filter) was reported and fixed. New releases have been made that contain bug fixes and improvements. For instance, the tool's performance has improved, and new types of mutations, such as mutating regular expressions, appeared.

2. Second, high integration effort. A system should be built on top of the mutation testing tool that automatically identifies changesets in pull requests and applies mutation testing based on that. For the experiments, files to mutate and tests to run were specified in the configuration file for each bug. However, in practice, this should be identified automatically, as developers would not want to spend additional time configuring the tool every time. The system should also display additional explanations for developers to explain the concept and required actions.

3. Third, the problem of equivalent mutants. The primary reason for automatically identifying equivalent mutants is that less developer time would be needed to consider equivalent mutants. While mutating only changed files in pull requests helps mitigate the problem of high computational cost, the problem of equivalent mutants remains. Equivalent mutants take a considerable amount of time to be reviewed, particularly if the same mutants are shown in every pull request concerning related files.

7.3 Using mutation testing in industry

While mutation testing has been considered a powerful testing technique, the experiments have demonstrated that it may not be suitable for every use case. It has been confirmed in this study that mutation score cannot be used as a reliable test quality metric; 0% score could mean there is only one unproductive mutant. However, mutants can still guide strengthening the test suite. In this case, for mutation testing to be useful, the mutants

must be coupled with real faults produced by the developers. If this is the case, then one has to consider how to alleviate the problems of computational expensiveness and developer time wasted on equivalent or generally unproductive mutants. One method to avoid computational expensiveness used in the experiments is reducing the number of mutants to the most relevant to the code change.

For a more efficient generation and usage of mutants, an analysis of the typical code patterns that contain bugs could be used along with the ongoing feedback from the developers to limit the number of mutants to the most actionable. In addition to that, one mutant per statement could be used to reduce the number of redundant mutants. Moreover, a set of change-aware mutants could be used rather than the mutants from the changed files. This means that *only* mutants relevant to the change and *all* mutants relevant to the change would be shown to the developers; these may come from different parts of the program and not only changesets and changed files. Other possible techniques to resolve the problems of mutation testing were described in Section 2.7.

However, if mutation testing is not beneficial for a particular use case, other ways to reduce the number of software bugs could be explored. Based on the experiments in this case study, more focus could be on the integration-level testing. Moreover, a complex logic that causes bugs could also indicate that the logic could be simplified or specifications could be more exhaustive. For example, Behavior-Driven Development could be leveraged for thorough specifications on which both business and technical sides could align. A more extreme way would be formal methods, mathematically rigorous techniques for designing and verifying software systems. Formal methods are commonly used in safety-critical software systems.

7.4 Limitations of the study

There are some limitations in this research that should be considered.

1. The first limitation is that only one mutation testing tool - Stryker4s - was used in the experiments. The reason for that is that this is the only active tool that exists for the Scala language. It is considered a limitation as research has shown that different mutation testing tools can lead to different results based on the implementation and available mutant operators.

2. The second limitation is that experiments on past software bugs have only been conducted on two systems. In addition to that, those are both Scala systems. The research has shown that different sets of mutants can be generated based on the language constructs, so the results might differ if applied to the services written in a different programming language.

3. The third limitation is the restriction mentioned above about the parts of the program to which apply mutation testing. Mutants relevant to the change might be in other parts of the program, not only changed files.

8 Conclusions

In this thesis, experiments with past software bugs of two Twilio software systems were conducted to determine whether mutation testing would have helped prevent them. In particular, the aim was to see whether the unit tests added during the mutation testing process would have helped to detect the bugs.

The results have shown that for the first service, in 1 out of 10 cases, mutation testing would have helped to identify one of the issues with an algorithm and reduce the number of fixes. For the second service, it was not helpful for any of the 7 bugs. Thus, mutation testing did not notably help to improve the fault detection capability of unit tests in the considered systems. Possible reasons for this could be the sources and level of complexity of the bugs. One suggestion is that mutation testing can help identify weakly tested pieces of code when applied to algorithms with complex logic. During the experiments, it was shown that mutation testing could also point out redundant test cases that do not bring any value and reveal inefficiencies in code.

However, there are some significant barriers to adopting mutation testing at the moment. They are immaturity of the tool, additional effort to set up a system that would automatically configure files and test suites to be used, and an abundance of equivalent mutants that waste valuable developer time. Ideally, the system should minimize the number of generated and shown mutants and explain the mutants to developers in an understandable way.

Adopting a new complex tool at the company level is an important decision that requires careful consideration and approval from both technical and business sides. The results of the experiments suggest that mutation testing does not provide enough benefits to outweigh all the drawbacks associated with it. Compared to other testing criteria, such as statement and branch coverage, it requires a lot more effort to be used, but the provided value does not create enough incentive for putting resources into it.

References

[1] Stryker mutator. `https://stryker-mutator.io/`. Last accessed: 2021-07-17.

[2] AGHAMOHAMMADI, A., AND MIRIAN-HOSSEINABADI, S.-H. The threat to the validity of predictive mutation testing: The impact of uncovered mutants. *arXiv preprint arXiv:2005.11532* (2020).

[3] AHMED, I., GOPINATH, R., BRINDESCU, C., GROCE, A., AND JENSEN, C. Can testedness be effectively measured? In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2016), pp. 547–558.

[4] ANDREWS, J. H., BRIAND, L. C., AND LABICHE, Y. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering* (2005), pp. 402–411.

[5] BAKER, R., AND HABLI, I. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *IEEE Transactions on Software Engineering 39*, 6 (2012), 787–805.

[6] BELLER, M., WONG, C.-P., BADER, J., SCOTT, A., MACHALICA, M., CHANDRA, S., AND MEIJER, E. What it would take to use mutation testing in industry—a study at facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (2021), IEEE, pp. 268–277.

[7] CHEKAM, T. T., PAPADAKIS, M., LE TRAON, Y., AND HARMAN, M. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (2017), IEEE, pp. 597–608.

[8] DARAN, M., AND THÉVENOD-FOSSE, P. Software error analysis: A real case study involving real faults and mutations. *ACM SIGSOFT Software Engineering Notes 21*, 3 (1996), 158–171.

[9] DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. *Computer 11*, 4 (1978), 34–41.

[10] DO, H., AND ROTHERMEL, G. A controlled experiment assessing test case prioritization techniques via mutation faults. In *21st IEEE International Conference on Software Maintenance (ICSM'05)* (2005), IEEE, pp. 411–420.

[11] DUQUE-TORRES, A., DOLIASHVILI, N., PFAHL, D., AND RAMLER, R. Predicting survived and killed mutants. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (2020), IEEE, pp. 274–283.

[12] FRANKL, P. G., WEISS, S. N., AND HU, C. All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software 38*, 3 (1997), 235–253.

[13] FRANKL, P. G., AND WEYUKER, E. J. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering 14*, 10 (1988), 1483–1498.

[14] GOPINATH, R., AHMED, I., ALIPOUR, M. A., JENSEN, C., AND GROCE, A. Does choice of mutation tool matter? *Software Quality Journal 25*, 3 (2017), 871–920.

[15] HEMMATI, H. How effective are code coverage criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security* (2015), IEEE, pp. 151–156.

[16] JIA, Y., AND HARMAN, M. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering 37*, 5 (2010), 649–678.

[17] JUST, R., JALALI, D., INOZEMTSEVA, L., ERNST, M. D., HOLMES, R., AND FRASER, G. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), pp. 654–665.

[18] KAKARLA, S., MOMOTAZ, S., AND NAMIN, A. S. An evaluation of mutation and data-flow testing: A meta-analysis. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops* (2011), IEEE, pp. 366–375.

[19] KINTIS, M., AND MALEVRIS, N. Using data flow patterns for equivalent mutant detection. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops* (2014), IEEE, pp. 196–205.

[20] KINTIS, M., PAPADAKIS, M., JIA, Y., MALEVRIS, N., LE TRAON, Y., AND HARMAN, M. Detecting trivial mutant equivalences via compiler optimisations. *IEEE Transactions on Software Engineering 44*, 4 (2017), 308–333.

[21] KINTIS, M., PAPADAKIS, M., PAPADOPOULOS, A., VALVIS, E., AND MALEVRIS, N. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2016), IEEE, pp. 147–156.

[22] KINTIS, M., PAPADAKIS, M., PAPADOPOULOS, A., VALVIS, E., MALEVRIS, N., AND LE TRAON, Y. How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering 23*, 4 (2018), 2426–2463.

[23] LAURENT, T., PAPADAKIS, M., KINTIS, M., HENARD, C., LE TRAON, Y., AND VENTRESQUE, A. Assessing and improving the mutation testing practice of pit. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)* (2017), IEEE, pp. 430–435.

[24] LE, D., ALIPOUR, M. A., GOPINATH, R., AND GROCE, A. Mutation testing of functional programming languages. *Oregon State University, Tech. Rep* (2014).

[25] LI, N., PRAPHAMONTRIPONG, U., AND OFFUTT, J. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *2009 International Conference on Software Testing, Verification, and Validation Workshops* (2009), IEEE, pp. 220–229.

[26] MA, W., LAURENT, T., OJDANIĆ, M., CHEKAM, T. T., VENTRESQUE, A., AND PAPADAKIS, M. Commit-aware mutation testing. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2020), IEEE, pp. 394–405.

[27] MA, Y.-S., OFFUTT, J., AND KWON, Y.-R. Mujava: a mutation system for java. In *Proceedings of the 28th international conference on Software engineering* (2006), pp. 827–830.

[28] NAMIN, A. S., AND KAKARLA, S. The use of mutation in testing experiments and its sensitivity to external threats. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (2011), pp. 342–352.

[29] OFFUTT, A. J., AND CRAFT, W. M. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability 4*, 3 (1994), 131–154.

[30] OFFUTT, A. J., AND PAN, J. Automatically detecting equivalent mutants and infeasible paths. *Software testing, verification and reliability 7*, 3 (1997), 165–192.

[31] OFFUTT, A. J., PAN, J., TEWARY, K., AND ZHANG, T. An experimental evaluation of data flow and mutation testing. *Software: Practice and Experience 26*, 2 (1996), 165–176.

[32] OFFUTT, A. J., AND UNTCH, R. H. Mutation 2000: Uniting the orthogonal. *Mutation testing for the new century* (2001), 34–44.

[33] PAPADAKIS, M., HENARD, C., HARMAN, M., JIA, Y., AND LE TRAON, Y. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (2016), pp. 354–365.

[34] PAPADAKIS, M., JIA, Y., HARMAN, M., AND LE TRAON, Y. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (2015), vol. 1, IEEE, pp. 936–946.

[35] PAPADAKIS, M., KINTIS, M., ZHANG, J., JIA, Y., LE TRAON, Y., AND HARMAN, M. Mutation testing advances: an analysis and survey. In *Advances in Computers*, vol. 112. Elsevier, 2019, pp. 275–378.

[36] PAPADAKIS, M., SHIN, D., YOO, S., AND BAE, D.-H. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2018), IEEE, pp. 537–548.

[37] PETROVIĆ, G., AND IVANKOVIĆ, M. State of mutation testing at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice* (2018), pp. 163–171.

[38] PETROVIĆ, G., IVANKOVIĆ, M., FRASER, G., AND JUST, R. Does mutation testing improve testing practices? In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (2021), IEEE, pp. 910–921.

[39] PETROVIĆ, G., IVANKOVIĆ, M., FRASER, G., AND JUST, R. Practical mutation testing at scale. *arXiv preprint arXiv:2102.11378* (2021).

[40] PETROVIC, G., IVANKOVIC, M., KURTZ, B., AMMANN, P., AND JUST, R. An industrial application of mutation testing: Lessons, challenges, and research directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (2018), IEEE, pp. 47–53.

[41] RAMLER, R., WETZLMAIER, T., AND KLAMMER, C. An empirical study on the application of mutation testing for a safety-critical industrial software system. In *Proceedings of the Symposium on Applied Computing* (2017), pp. 1401–1408.

[42] ZHANG, J., ZHANG, L., HARMAN, M., HAO, D., JIA, Y., AND ZHANG, L. Predictive mutation testing. *IEEE Transactions on Software Engineering 45*, 9 (2018), 898–918.

Appendix

I. Examples of test cases

```
"AZ/Full Country offer" - {
  "should create but not override existing fallbacks in vendor offer" in {
    val pipeline = spy(createPipeline())
    val offer = PipelineTest.smsRateData.copy(offerType = SmsOfferType.VendorOffer.toString)
    val firstRate = SmsOffer.offerToDbModel(offer).head.copy(ratedeckCategory = AZ,
      mcc = "XX", effectiveFrom = effectiveFrom.toInstant, forecastRate = None)

    val expectedRates = List(
      RateForMccMnc("XX", "001", effectiveFrom, BigDecimal.valueOf(0.15), None),
      RateForMccMnc("XX", "002", effectiveFrom, BigDecimal.valueOf(0.25), None)
    )
    val computedFallbacks = Vector(RateForMccMnc("XX", "999", effectiveFrom, BigDecimal.valueOf(0.25), None))
    doReturn(computedFallbacks).when(pipeline).computeCountryFallbacks(expectedRates, overrideFallback = false)

    val azRates = List(
      firstRate.copy(mnc = "001", rate = Option(BigDecimal.valueOf(0.15))),
      firstRate.copy(mnc = "002", rate = Option(BigDecimal.valueOf(0.25)))
    )
    pipeline.fallbacks(azRates).futureValue shouldBe computedFallbacks

    val fullCountryRates = azRates.map(_.copy(ratedeckCategory = FullCountry))
    pipeline.fallbacks(fullCountryRates).futureValue shouldBe computedFallbacks
  }
```

Figure 13. Existing incorrect test case for Mutants 2 and 3

```
"AZ/Full Country offer" - {
  "should create but not override existing fallbacks in vendor offer" in {
    val pipeline = spy(createPipeline())
    val offer = PipelineTest.smsRateData.copy(offerType = SmsOfferType.VendorOffer.toString)
    val firstRate = SmsOffer.offerToDbModel(offer).head.copy(ratedeckCategory = AZ,
      mcc = "XX", effectiveFrom = effectiveFrom.toInstant, forecastRate = None)

    val expectedRates = List(
      RateForMccMnc("XX", "001", effectiveFrom, BigDecimal.valueOf(0.15), None),
      RateForMccMnc("XX", "002", effectiveFrom, BigDecimal.valueOf(0.25), None)
    )
    val computedFallbacks = Vector(RateForMccMnc("XX", "999", effectiveFrom, BigDecimal.valueOf(0.25), None))
    doReturn(computedFallbacks).when(pipeline).computeCountryFallbacks(expectedRates, overrideFallback = false)

    val azRates = List(
      firstRate.copy(mnc = "001", rate = Option(BigDecimal.valueOf(0.15))),
      firstRate.copy(mnc = "002", rate = Option(BigDecimal.valueOf(0.25))),
      firstRate.copy(mnc = "999", rate = Option(BigDecimal.valueOf(0.2)))
    )
    pipeline.fallbacks(azRates).futureValue shouldBe List()

    val fullCountryRates = azRates.map(_.copy(ratedeckCategory = FullCountry))
    pipeline.fallbacks(fullCountryRates).futureValue shouldBe List()
  }
```

Figure 14. Corrected test case for Mutants 2 and 3

```
"AZ/Full Country offer" - {
  "should create and override fallbacks in non-vendor offer" in {
    val pipeline = spy(createPipeline())
    val offer = PipelineTest.smsRateData.copy(offerType = SmsOfferType.Dashboard.toString)
    val firstRate = SmsOffer.offerToDbModel(offer).head.copy(ratedeckCategory = AZ,
      mcc = "XX", effectiveFrom = effectiveFrom.toInstant, forecastRate = None)

    val expectedRates = List(
      RateForMccMnc("XX", "001", effectiveFrom, BigDecimal.valueOf(0.15), None),
      RateForMccMnc("XX", "002", effectiveFrom, BigDecimal.valueOf(0.25), None)
    )
    val computedFallbacks = Vector(RateForMccMnc("XX", "999", effectiveFrom, BigDecimal.valueOf(0.25), None))
    doReturn(computedFallbacks).when(pipeline).computeCountryFallbacks(expectedRates, overrideFallback = true)

    val azRates = List(
      firstRate.copy(mnc = "001", rate = Option(BigDecimal.valueOf(0.15))),
      firstRate.copy(mnc = "002", rate = Option(BigDecimal.valueOf(0.25)))
    )
    pipeline.fallbacks(azRates).futureValue shouldBe computedFallbacks

    val fullCountryRates = azRates.map(_.copy(ratedeckCategory = FullCountry))
    pipeline.fallbacks(fullCountryRates).futureValue shouldBe computedFallbacks
  }
```

Figure 15. Existing incorrect test case for Mutant 4

```
"AZ/Full Country offer" - {
  "should create and override fallbacks in non-vendor offer" in {
    val pipeline = spy(createPipeline())
    val offer = PipelineTest.smsRateData.copy(offerType = SmsOfferType.Dashboard.toString)
    val firstRate = SmsOffer.offerToDbModel(offer).head.copy(ratedeckCategory = AZ,
      mcc = "XX", effectiveFrom = effectiveFrom.toInstant, forecastRate = None)

    val expectedRates = List(
      RateForMccMnc("XX", "001", effectiveFrom, BigDecimal.valueOf(0.15), None),
      RateForMccMnc("XX", "002", effectiveFrom, BigDecimal.valueOf(0.25), None)
    )
    val computedFallbacks = Vector(RateForMccMnc("XX", "999", effectiveFrom, BigDecimal.valueOf(0.25), None))
    doReturn(computedFallbacks).when(pipeline).computeCountryFallbacks(expectedRates, overrideFallback = true)

    val azRates = List(
      firstRate.copy(mnc = "001", rate = Option(BigDecimal.valueOf(0.15))),
      firstRate.copy(mnc = "002", rate = Option(BigDecimal.valueOf(0.25))),
      firstRate.copy(mnc = "999", rate = Option(BigDecimal.valueOf(0.1)))
    )
    pipeline.fallbacks(azRates).futureValue shouldBe computedFallbacks

    val fullCountryRates = azRates.map(_.copy(ratedeckCategory = FullCountry))
    pipeline.fallbacks(fullCountryRates).futureValue shouldBe computedFallbacks
  }
```

Figure 16. Corrected test case for Mutant 4

```
"Partial offer" - {
  "should not return a new fallback if it is equivalent to existing fallback" in {
    val smsRatesDao = mock[SmsRatesDao]
    val enrichedRates = mock[EnrichedSmsRates]
    val pipeline: SmsPipeline = spy(createPipeline().copy(smsRates = smsRatesDao, enrichedSmsRates = enrichedRates))

    val offer = PipelineTest.smsRateData

    val firstRate = SmsOffer.offerToDbModel(offer).head.copy(ratedeckCategory = Partial,
      mcc = "XX", effectiveFrom = now.toInstant, forecastRate = None)

    val partialRates = List(
      firstRate.copy(mnc = "001", rate = Option(BigDecimal.valueOf(0.15))),
    )
    val existingRates = List(
      firstRate.copy(mnc = "001", rate = Option(BigDecimal.valueOf(0.2))),
      firstRate.copy(mnc = "999", rate = Option(BigDecimal.valueOf(0.15))),
    )

    doReturn(FastFuture.successful(existingRates)).when(smsRatesDao)
      .listActiveRatesTillSpecificTime(firstRate.providerSid, firstRate.effectiveFrom)

    pipeline.fallbacks(partialRates).futureValue shouldBe Vector.empty
  }
```

Figure 17. Added test case for Mutant 9

```
"Partial offer" - {
  "should create fallbacks only for mccs that do not have it in the ratedeck" in {
    val smsRatesDao = mock[SmsRatesDao]
    val enrichedRates = mock[EnrichedSmsRates]
    val pipeline: SmsPipeline = spy(createPipeline().copy(smsRates = smsRatesDao, enrichedSmsRates = enrichedRates))

    val offer = PipelineTest.smsRateData

    val firstRate = SmsOffer.offerToDbModel(offer).head.copy(ratedeckCategory = Partial,
      mcc = "XX", effectiveFrom = now.toInstant, forecastRate = None)

    val partialRates = List(
      firstRate.copy(mnc = "001", rate = Option(BigDecimal.valueOf(0.15))),
      firstRate.copy(mnc = "999", rate = Option(BigDecimal.valueOf(0.1))),
      firstRate.copy(mcc = "YY", mnc = "001", rate = Option(BigDecimal.valueOf(0.15))),
      firstRate.copy(mcc = "ZZ", mnc = "999", rate = Option(BigDecimal.valueOf(0.25)))
    )

    val existingRates = List(
      firstRate.copy(mnc = "999", rate = Option(BigDecimal.valueOf(0.2)))
    )

    val computedFallbacks = Vector(RateForMccMnc("YY", "999", now, BigDecimal.valueOf(0.15), None))

    doReturn(FastFuture.successful(existingRates)).when(smsRatesDao)
      .listActiveRatesTillSpecificTime(firstRate.providerSid, firstRate.effectiveFrom)

    pipeline.fallbacks(partialRates).futureValue shouldBe computedFallbacks
  }
```

Figure 18. Added test case for Mutant 7

II. Mutation testing report

```
private[pipelines] def fallbacks(newRates: Seq[RateSms]): Future[Seq[RateForMccMnc]] = {
  newRates.headOption
    .map(firstRate => {
     val addedRates = newRates.flatMap(_.asMccMncRate)
     firstRate.ratedeckCategory match {
       case AZ | FullCountry if firstRate.offerType == VendorOffer =>
         FastFuture.successful(computeProviderFallbacks(addedRates, Vector.empty, overrideFallback = false))
       case AZ | FullCountry =>
         FastFuture.successful(computeProviderFallbacks(addedRates, Vector.empty, overrideFallback = true))
       case Partial =>
         if (?newRates.forall(_.mnc == FallbackRateNetwork)) {
           FastFuture.successful(Vector.empty)
         } else {
           val insertedCountryCodes = newRates.map(_.mcc).toSet
           smsRates
             .listActiveRatesTillSpecificTime(firstRate.providerSid, firstRate.effectiveFrom)
             .map(_.toList)
             .map(_.filter(rate => insertedCountryCodes.contains(rate.mcc)))
             .map(_.flatMap(_.asMccMncRate))
             .map(computeProviderFallbacks(addedRates, _, overrideFallback = true))
         }
     }
    })
    .getOrElse(FastFuture.successful(Vector.empty))
}
```

Figure 19. Mutation Testing Report after the test cases were added

III. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Gaukhar Dauzhan**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Mutation Testing for Improving Fault Detection Capability of Unit Tests: A Case Study,

   supervised by Dietmar Pfahl and Eerik Potter.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Gaukhar Dauzhan
01/08/2021