Prompt

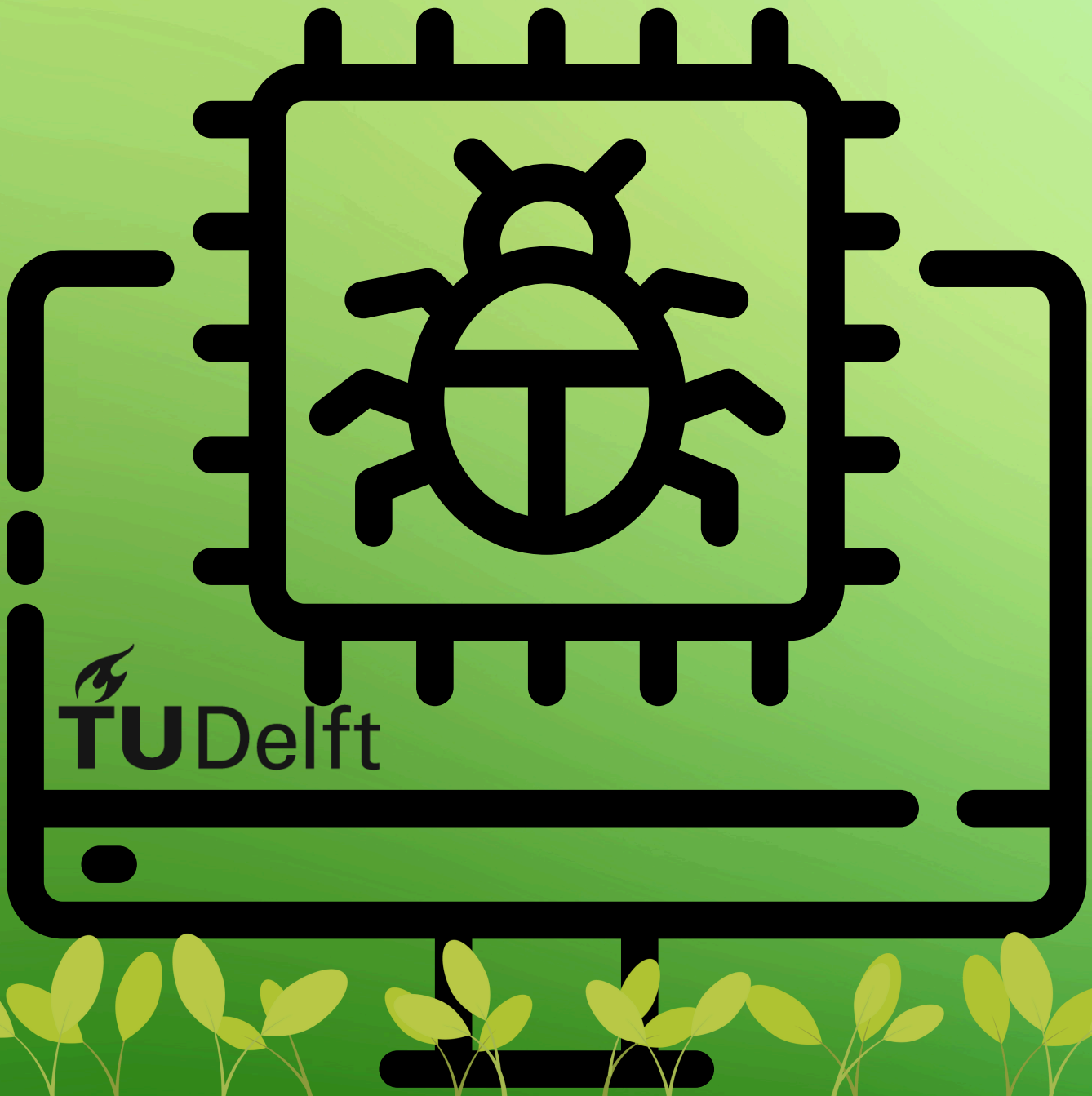
Seed

Generate

Seeding For Test Case Generator with LLMs

Sergey Datskiv

Delft University Of Technology



Prompt, Seed, Generate: Seeding For Test Case Generator with LLMs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Sergey Datskiv

Cyber Security Research Group
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

JetBrains
Gelrestraat 16 Amsterdam
Amsterdam, the Netherlands
https://lp.jetbrains.com/research/software-testing/

Cover picture: Created on Canva by Sergey Datskiv from freely available elements.

Prompt, Seed, Generate: Seeding For Test Case Generator with LLMs

Abstract

Unit test case generation aims to help software developers test programmes. The evolutionary algorithm is one of the successful approaches for unit test case generation that evolves problem solutions over time. Previous research on seeding, the use of previously available information to improve search performance, showed positive improvements in unit test case generation. However, this approach cannot be used in the absence of previously available information, such as existing unit test cases.

The recent increased availability of Large Language Models (LLMs), which were trained on various corpora of previously available data, provides an opportunity to address the seed absence problem. We devised an approach involving TestSpark and EvoSuite to see the impact of LLM-based seeding on unit test case generation. TestSpark, an IntelliJ plugin, uses ChatGPT-4o to generate test cases which we later supply as a seed for EvoSuite's seeding strategies such as cloning and carving. We evaluated our approach on a set of 136 Java 11 projects from the GitBug-Java dataset w.r.t. line coverage, branch coverage, mutation score, and area under the curve.

Our results show that LLM-based seeding has the potential to improve EvoSuite's unit test case generation if it manages to extract information from the seed. Our approach experiences significant struggles to supply LLM-generated tests from which information can be extracted. We lost 63% of the benchmark classes because LLM did not generate functional tests for all experiment iterations. Meanwhile, another 24% of the benchmarks are excluded because EvoSuite seeding does not extract any information from them.

Thesis Committee:

Chair:	Prof. Dr. G. Smaragdakis, Faculty EEMCS, TU Delft
University supervisor:	Dr. A. Panichella, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Voulimeneas, Faculty EEMCS, TU Delft
Company supervisor:	Dr. P. Derakhshanfar, Software Testing Research Team, JetBrains
Company supervisor:	S. Lukasczyk, Software Testing Research Team, JetBrains

Preface

There are many people who I want to thank for being part of my journey to this point in my life. Firstly, I would like to thank TU Delft and JetBrains and the people representing them, without whom this thesis would not have been possible. TU Delft is the university that taught me my B.Sc. and M.Sc. of Computer Science and provided an opportunity to meet many great people. I would like to thank Annibale for being my supervisor and for his time, support, and guidance. George and Alex for being the members of my thesis committee. Meanwhile, JetBrains is not just a company which provided me with many valuable tools for many of my Computer Science projects, but they were also involved in this project as a part of the TU Delft-JetBrains collaboration. Pouria and Stephan are my JetBrains supervisors, whom I also want to thank for all their time, support, and guidance. Thank you for being my supervisors and friends!

Secondly, I would like to thank all my friends from the university and outside it for being with me throughout all these years. You have helped me a lot through various personal and academic struggles, but it would also not have been fun without you. Thank you for being my friends!

Lastly, I would like to thank my entire family for all the opportunities they provided me with to reach this point in my life. I would not be where I am today without your influence on my life, and I am very grateful to have such a wonderful and supportive family. Thank you for being my family!

Sergey Datskiv
June 20, 2025

Contents

List of Figures

Chapter 1

Introduction

Software testing is an important part of the software development cycle, as it helps to discover bugs and possible problems arising due to accidental or intended events. It is a process through which we can gain more confidence that the developed software will perform as intended and will not be compromised in various scenarios. However, high-quality testing can be a time- and resource-intensive process, as it requires the tester to correctly arrange the testing environment, execute the code under test in that specific environment, and assert its results. To reduce the burden placed on testers, researchers have created and improved many automated testing tools [2, 8, 12, 19, 31, 35, 38, 48] over the decades [6]. Those tools were able to achieve notable results in the area of code coverage in competitions [17, 22, 24, 25, 35, 43], and some studies even show their ability to find real-world faults [5, 23]. The search-based approach to software testing is one of the most well-established and effective approaches [17, 24, 25, 35, 43]. However, it also has its limitations and areas of improvement, such as the creation of more complex objects and the improvement of assertions [50]. The emergence of Large Language Models (LLMs) and their quick adaptation in the area of software development has sparked large interest in the possibility of using LLMs to improve previously known approaches, test case generation is no exception.

For decades, the most prevalent approaches in the test case generation community were random search techniques, search-based (SBST) techniques, and symbolic execution-based techniques [6]. Each approach has its own advantages and disadvantages, which have been studied over time. The evolutionary algorithm (EA) belonging to the evolutionary computation (EC) community is an often used base for building search-based algorithms for unit test case generation. EA consists of four stages: initialization of a population that represents solutions to the problem, creation of offsprings through random variation, fitness evaluation of the solutions, and selection of solutions for the next iteration. The main idea behind EA is to improve the fitness of the population through different iterations of the search with the help of a specific fitness function which guides the population to a certain goal over time. Both communities, EC and SBST, have explored various ways to improve the performance of their algorithms in finding suitable solutions. One of such research areas was seeding, a technique that focuses on using previously available knowledge, such as existing solutions, to benefit the ongoing search process. Seeding techniques were implemented in SBST ap-

proaches and studied. For example, Fraser and Arcuri and Rojas et al. implemented seeding strategies within EvoSuite [19], a unit test case generation tool for Java-based projects. Their studies [21, 46] showed that seeding can make a significant difference in the performance of a test case generation tool and explored different seeding strategies, such as cloning and carving. The main idea behind cloning and caving seeding strategy is to extract information from already existing tests and, at times, insert them into the EA's search process in hopes of helping it. However, in the absence of previous information, these seeding techniques are of no use [50].

With the recent increase in availability of LLMs, which were trained on various corpora of previously available data, an interesting question arises; "in the absence of prior knowledge, can LLMs be used to provide the missing seed information and thus improve the performance of the test generation algorithm?"

This is the research question which we try to address in this thesis. That is, "**What is the impact of ChatGPT-4o generated tests when supplied as a seed for EvoSuite's carving and cloning strategies on test case generation?**"

Answering this research question should provide insight into how LLMs can be introduced in existing EA-based unit test case generation tools such as EvoSuite and what kind of impact LLM-based seed could have on unit test case generation.

Our hypothesis is that the search can reuse the information contained in the seed without having to discover this information by itself. By reusing the existing information, the search is able to dedicate more iterations to working on the yet-to-be-uncovered set of objectives. This should translate into better line coverage and branch coverage, or in the case of convergence to the same coverage, there should be a faster convergence to the final number, which can be measured through the area under the curve. Additionally, existing test cases might have better inputs for catching bugs, which would be reflected in the mutation score.

The exact contributions of the thesis are in the research carried out to answer the research question, the modifications to the EvoSuite & TGA-Pipeline[1], and a replication package available on Zenodo with the following doi `10.5281/zenodo.15698634`.

TGA-Pipeline is a pipeline produced for paper [3] by Abdullin et al. that simplifies the process of evaluating tools such as EvoSuite. In our research, we use this pipeline to carry out our evaluation and make the appropriate changes[2] to make this happen, e.g. include the ability to supply existing test cases and fix several Windows platform-specific bugs.

The remaining parts of the thesis are split into seven chapters. Chapter 2 will inform you about the background information required to understand the thesis. It will be followed by a Chapter 3 where we overview the related literature. Next, in Chapter 4 we explain our proposed idea for LLM-based seeding in the EvoSuite tool. In Chapter 5, having previously explained our idea and its implementation, we move onto the empirical evaluation. Chapter 6 presents the results and follows with a discussion of them. Lastly, the thesis ends with Chapter 7 stating the conclusion and suggesting future work.

[1] Link to the original repo: https://github.com/plan-research/tga-pipeline

[2] Link to our fork of the pipeline. https://github.com/SergeyDatskiv/tga-pipeline/tree/SergeyDatskiv/development

Chapter 2

Background

2.1 Software Testing Metrics

Software testing is an important aspect of software development because it helps us identify the presence of problems in the program under test [7]. However, as the complexity, functionality, and size of the program under test expands, so does the difficulty associated with testing that particular program and knowing if it is tested well [7]. A good test suite should be able to cover many, if not all, parts of a program and indicate when something is wrong. To help determine the quality of a test suite, researchers and developers turn to various metrics which act as proxies to measure the effectiveness of test suites [60]. For example, a metric that is often used to inform us how much of a program was covered by execution of a test suite is code coverage [60]. However, code coverage does not inform testers if the test can reveal faults in the program. To evaluate the quality of a test suite to reveal faults, we need a metric that attempts to measure whether the assertions of a test case can detect faults [44]. One of the most commonly used metrics for this is the mutation score. However, we should remember that testing cannot prove the correctness, as Edsger W. Dijkstra said it, "Program testing can be used to show the presence of bugs, but never to show their absence!"

2.1.1 Structured Code Coverage

Structured code coverage metrics indicate the amount of code executed by a given test or set of tests. This indication can be given on different levels of granularity, from statement to path coverage. The lowest level is the coverage of statements / lines, where we simply report the number of statements / lines executed compared to the total number of statements/lines [34]. Then there is branch coverage, which counts the number of executed branch paths compared to the total possible branches [60]. Branch coverage also subsumes line/statement coverage, meaning that if we have 100% branch coverage, then we also have 100% statement/line coverage [1]. Regardless of the chosen code coverage metric, its result split the program into two parts, covered and not covered, thereby informing the testers of places where the bugs might still be hiding because those parts of code were never reached during current test execution.

```
1 public static int addition(int x, int y)
2 {
3     return x * y;
4 }
```

Figure 2.1: Example of an incorrect addition function. Highlighted is the statement covered by the test in Figure 2.2.

```
1 @Test
2 public void TestAddTwoAndTwo()
3 {
4     int x, y = 2;
5     Calculator.addition(x, y);
6 }
```

Figure 2.2: Example of a test achieving full statement coverage. Covered statements are highlighted in Figure 2.1.

Figure 2.3: Example of an incorrect addition function in Java (Figure 2.1) and a test achieving full statement coverage (Figure 2.2). Green highlighting shows the function's statement coverage achieved with the given test.

Unfortunately for testers, having high code coverage, i.e. reaching all parts of the program does not give any guarantees that there are no bugs. This is because some bugs might require specific conditions for them to occur and become visible. For example, consider a function from Figure 2.1 that adds two numbers together. The function has a typo where a multiplication is used instead of an addition leading to a correct result in some cases such as $2+2=4$ since $2*2=4$, but not in others. A single test, such as the one in Figure 2.2, will be enough to achieve full coverage of the statements for this function; however, depending on the test, it might not reveal the problem. Furthermore, as you might have noticed from the example in Figure 2.3, the code coverage metric does not account for the presence or absence of assertions in the tests. That is, even if the bug is found and triggered, it does not necessarily mean that the test suite caught it. Assertions are a vital part of a test, and their quality is better captured by the mutation score than by coverage metrics. Nevertheless, we still use code coverage metrics like line and branch coverage in practice because we surely will not be able to find bugs in parts of the programme which tests never execute. Furthermore, Kochhar et al. showed in [27] that there is a positive correlation between the coverage metric and the effectiveness of bug kill of a test suite, indicating the value of the code coverage metric.

2.1.2 Mutation Score

As mentioned above in Section 2.1.1, the code coverage metrics do not directly represent the fault-detection capability of the tests. Hence, there is a need for another testing strategy and a metric which would be more informative in that regard. One such metric is a mutation score that is obtained through mutation testing. Mutation testing is the process of injecting artificial defects into the source code and then executing the test suite to see if the tests catch those artificial defects [44]. There are many artificial defects which can be introduced into the program under test. The typical examples include changes of various operators, such as arithmetic, logic, relational, and more. The defects are supposed to mimic the real mistake

that could take place during program development, such as incorrectly placing the value of "less than or equal ($\leq$)" value instead of "less than ($<$)" or the one depicted in Figure 2.6 where the original code changes from the equality check, Figure 2.4, to the inequity in Figure 2.5. After inserting the mutants, the test suite is executed again, and if we can see a change in the test states, from passing to failing, then we can say that the defect was caught.

```
1 ...
2 if (a == b) {
3     if (b == c) {
4         return "EQUILATERAL";
5     } else return "ISOSCELES";
6 } else ...
```

Figure 2.4: A snippet of a original not mutated function.

```
1 ...
2 if (a == b) {
3     if (b != c) {
4         return "EQUILATERAL";
5     } else return "ISOSCELES";
6 } else ...
```

Figure 2.5: Code snippet with the injected mutant (== $\rightarrow$!=) at line 3.

Figure 2.6: Example of artificial defect injection for mutation testing. Figure 2.4 depicts the original not mutated function, while Figure 2.5 depicts how the original function could be changed after injection of an artificial defect, aka. mutant, bug.

2.2 Automated Test Case Generation

Automated test case generation is a mature field in software testing [6] which has already produced many different tools [2, 8, 12, 19, 31, 35, 38, 48] using different approaches to test case generation. One of the tools, EvoSuite [19, 42], which iteratively won competitions [24, 25, 35, 43] led to a genetic algorithm as its main approach to generate test cases. The genetic algorithm is a subclass of evolutionary algorithms that are inspired by Darwin's theory of evolution and principles of natural selection [18].

2.2.1 Evolutionary Algorithm Overview

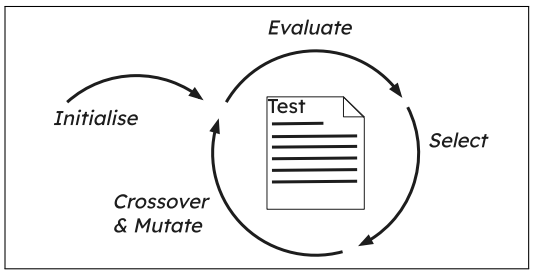


Figure 2.7: Diagram depicting the cyclic process of an evolutionary algorithm along with its key stages.

Darwin's work on evolution is the motivation for the evolutionary algorithm (EA), as it leverages concepts such as natural selection and survival of the fittest [18]. An EA algorithm

usually has four stages, namely: initialization, offspring creation, evaluation, and selection. Figure 2.7 visualizes the algorithm and its four stages.

On a high level, the algorithm starts with an initial population, possibly a random one, and a problem-specific fitness function that determines the fitness of an individual. We select a number of the fittest individuals[1] and exchange their genetic information in hopes of producing a fitter offspring population. The iteration of this process should produce better performing individuals and will stop once a termination condition is reached.

To apply an EA to a problem, we need to find a way to represent a solution to that problem in the form of an individual whose genes (parameters) could be exchanged with another solution through random variation operations such as crossover and mutation [18]. Furthermore, we need to define a problem-specific fitness function to numerically identify people closer to achieving the desired goal [18].

2.2.2 Evolutionary Algorithm For Test Case Generation

As mentioned in section 2.2.1, to apply evolutionary algorithms to a problem, we need to find an encoding for the problem's solutions and formulate a function to measure the fitness of each created solution. There are different ways to formulate the test case generation problem [20, 41], but one of the most successful, as shown in competitions [24, 25, 35, 43], was in Panichella et al. as a multi-objective optimisation problem [41].

The idea behind Panichella et al.'s many-objective formulation of the problem is that "*test case generation is intrinsically a multi-objective problem, since the goal is covering multiple test targets (e.g., branches)*" [42]. For example, suppose that we chose line coverage as a metric by which we assess the test suite. If you recall section 2.1.1, then you remember that for line coverage we need to count the total number of lines reached when a test is executed and divide it by the total number of lines in a program under test. In such a case, each line of a program under test becomes our target and each test case becomes a possible solution because it has the potential to cover those targets. Thus, test case generation is formulated in a way where a test case is an individual of a population, and fitness functions measure the distance to covering a certain minimal target, e.g. line or branch.

In an evolutionary algorithm, a single individual in a population can be considered as a solution to the problem. A single test is an individual of a population, as can be seen in Figure 2.8. A test case can be represented as a list of statements, where each statement is a piece of executable code. This list of statements is what makes up the test and can be thought of as the DNA of this individual. This encoding of a test case as a list of statements allows us to apply several different genetic operations to it, such as selection, crossover, and mutation.

Selection is the process by which the algorithm chooses suitable individuals from the current population to produce the offspring, thereby, we hope, guiding the algorithm to better solutions. Although there are different selection operators, one of which is shown in Figure 2.9, all serve the same purpose. The operator pairs the selected individuals (test cases) and prepares to produce an offspring (new test case) based on the parent's DNA (test case's statement list) through the crossover operation.

[1] A hyper-parameters of an EA algorithm.

Figure 2.8: Example of a genetic algorithm population which is a list of test cases, i.e. a test suite where each individual is a single test case. A test case is a list of statements.

Crossover is a mixing of two DNAs. More concretely, given two lists of statements (DNA of two different tests), we can create a new list of statements (DNA of a new test). The example of the process can be found in Figure 2.13. The new list of statements, Figure 2.12, is a partial copy of the original first test up to a certain random point (Figure 2.10), and the remaining part is the copy of the second test up to the end (Figure 2.11). Similarly, for the selection operator, there are different crossover approaches, each with its own benefit. The crossover operation is usually applied to produce new test cases, which are also known as offsprings of a current, parent, population. Due to the usage of existing DNA to produce offspring, population diversity is negatively affected, and a mutation operation is applied to mitigate this consequence [4].

Mutation operations are changes performed to the DNA (list of statements) of an offspring to diversify them from the parent population. There are different ways in which it can be implemented. An example is shown in Figure 2.16, where Figure 2.14 shows the original solution (test case), while Figure 2.15 shows a possible outcome after mutation. Diversity is an important concept in an evolutionary algorithm because it helps escape the local optimum [4]. Mutation operations aim to change the values of the inherited offspring DNA through random modifications such as adding new statements, removing old statements, or changing existing ones. However, a high mutation rate should be avoided to prevent the search from becoming random due to the loss of good genes.

It should be noted that for each one of the genetic operations there exist several approaches to how it can be implemented, a nice overview is given by Alhijawi and Awajan in [4]. Due to the existence of several different approaches, genetic operations can be thought of as hyperparameters of an EA; however, the study [10] by Arcuri and Fraser showed that in the area of test case generation the suggested default parameters perform well and further

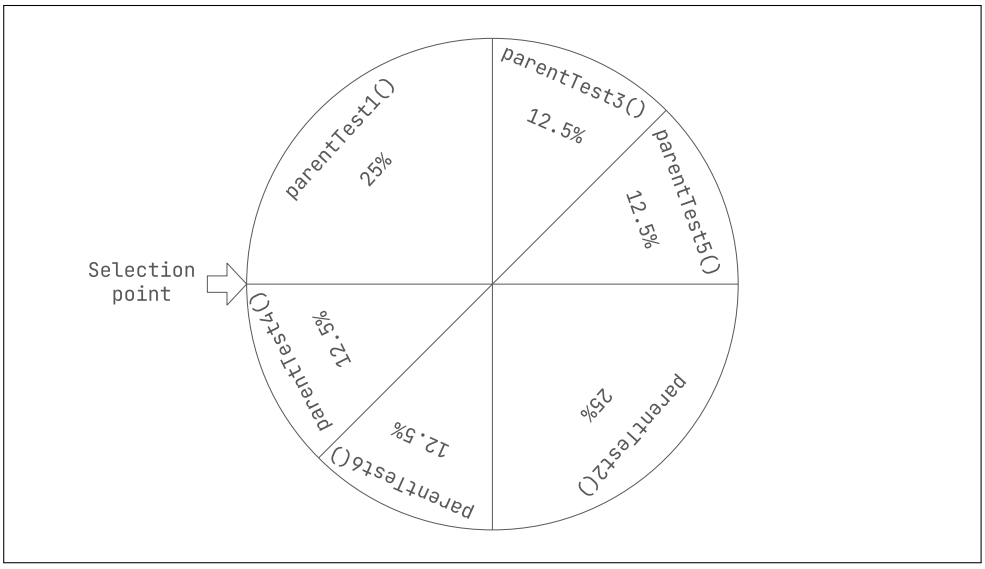


Figure 2.9: Example of Roulette Wheel selection (a fitness proportionate selection operator) with six parents two of which have 25% shares each while the remaining four have 12% wheel share each. The shares are calculated basaed on the individuals fitness score in comparison to others. Parent selectiton happens by spinning the roulette wheel to see which individuals lands on the selection point. The wheel is spinned twice as many times as the desired number of offspring individuals since a single offspring requires two parents.

tuning comes at a high cost for little improvement.

The second important part to creating an EA is the ability to evaluate the performance of each individual in the population, i.e. assess how well the test case does in covering the desired objectives. For that, we need a fitness function that can evaluate the test with respect to the desired target. This fitness function is different for each objective of a different type (e.g., line coverage, branch coverage, etc.) and ideally is supposed to guide the selection process to better individuals. For example, branch coverage is decomposed into multiple targets (objectives) where a single target is a possible branch path of a program. Branch distance formula, $f(x) = apl(P(x),t) + nbd(P(x),t)$, for a coverage target t with respect to a generated test case x is a sum of approach level ($apl(P(x),t)$) and normalized branch distance ($nbd(P(x),t)$), where $P(x)$ is an execution trace of a test case x ran on the program under test $P()$. The approach level gives us the minimum number of control nodes between an executed part and the coverage target. To get an approach level value, we identify control nodes (branches) which need to be covered by the execution path to get to the target. We then see which is the furthest control node reached by the execution path and count how many more nodes we need to get to the target node. The normalized branch distance is a formula specific to each type of branch condition, for more information and examples of how to compute the branch distance you can see work [42] by Panichella et al. among many others that covered that topic.

```
1  @Test
2  public void parentTest1() {
3      int side1 = 10;
4      int side2 = 10;
5      int side3 = 10;
6      String result;
7      result = determineTriangle(side1,
8                                 side2,
9                                 side3);
10 }
```

Figure 2.10: A test case selected from a parent population of a genetic algorithm for the crossover function. Highlighted parts indicate which lines will be copied to an offspring test case, Figure 2.12, as part of a single-point crossover operation.

```
1  @Test
2  public void parentTest2() {
3      int side1 = 10;
4      int side2 = 15
5      int side3 = 8;
6      String result;
7      result = determineTriangle(side1,
8                                 side2,
9                                 side3);
10 }
```

Figure 2.11: A different test case selected from a parent population of a genetic algorithm for the crossover function. Highlighted parts indicate which lines will be copied to an offspring test case, Figure 2.12, as part of a single-point crossover operation.

```
1  @Test
2  public void offspringTest() {
3      int side1 = 10;
4      int side2 = 10;
5      int side3 = 8;
6      String result;
7      result = determineTriangle(side1, side2, side3);
```

Figure 2.12: An example of a produced offspring test based on a single-point crossover operation between two parent test cases from Figures 2.10 and 2.11.

Figure 2.13: An example of a single-point crossover operation producing an offspring, Figure 2.12, based on two selected parents from Figures 2.10 and 2.11. Highlighted parts of the parent's figures indicate which lines from which parents are copied into an offspring.

2.2.3 Seeding

Evolutionary algorithms can start with a random population, i.e. a set of randomly created test cases where the list of statements was created by randomly adding new statements to it. We evaluate those test cases to understand which ones happened to perform better than others and afterwards apply genetic operations such as crossover and mutations, which are random in nature, to hopefully produce better test cases. This process of evaluating new test cases, picking the best performing ones, and applying genetic operators to produce better individuals is repeated until termination either by completion or out of search budget. The process is pretty random, but thanks to the evaluation step, it should be making progress towards better test candidates. The starting point of the algorithm, seed, can play an important role in determining the success of an algorithm to produce good tests as was shown by Fraser and Arcuri in [21]. Naturally, this question of seeding became a topic of interest for

```
1  @Test
2  public void originalOffspringTest() {
3      int side1 = 10;
4      int side2 = 10;
5      int side3 = 8;
6      String result;
7      result = determineTriangle(side1,
8                                 side2,
9                                 side3);
10 }
```

Figure 2.14: An example of an offspring test case generatetd through a crossover operation between two selected parents.

```
1  @Test
2  public void muatedOffspringTest() {
3      int side1 = 10;
4      int side2 = -1;
5      int side3 = 8;
6      String result;
7      result = determineTriangle(side1,
8                                 side2,
9                                 side3);
10 }
```

Figure 2.15: A mutated version of an offspring test case. Highlighted is the mutation which changed `int side2` from 10 to -1.

Figure 2.16: An example of a mutation operation performed on an offspring test case in Figure 2.14, the outcome of mutation is presented in Figure 2.15

```
1  public static String determineTriangle(int a, int b, int c) {
2      if (a == b) {
3          if (b == c) {
4              return "EQUILATERAL";
5          } else return "ISOSCELES";
6      } else {
7          if (a == c) {
8              return "ISOSCELES";
9          } else {
10             if (b == c) {
11                 return "ISOSCELES";
12             } else {
13                 return "SCALENE";
14             }
15         }
16     }
17 }
```

Figure 2.17: An example of a function under test. `determineTriangle` function takes three integer inputs one for each supposed side of a triangle, compares them, and returns the type of triangle.

researchers, and they attempted to improve seeding by incorporating previously available knowledge into the search process through different strategies. Those seeding strategies can be broadly split into two categories, extracting information from code under test or from other sources. The former involves approaches such as static, dynamic, and type seeding, among many others, while the latter includes but is not limited to things like using already generated tests as a seed. Rojas et al. explored the applicability of those strategies on test case generation in EvoSuite in [46].

Static seeding, as tried by Rojas et al., involves the idea of extracting static values from

the code under test in the hope that they can be useful to cover certain goals. It could take several iterations for the genetic algorithm to arrive at the matching value because, despite the guiding fitness function, the operations we perform on the individuals are random in nature. The algorithm won't know if the change is beneficial until it evaluates the change. However, what if instead of picking random values, we first analyse the source code and collect all static values like strings and integers into separate pools of data from which our algorithm can draw upon when needed. This would mean that instead of having to perform many random genetic operations, our algorithm might luckily get the correct value from the existing static pool of values, potentially right at the beginning of the search process.

Dynamic seeding follows the same idea as static seeding, but accounts for the situations where the algorithm might not know the exact values of a program under test until it executes it fully. We have a branch condition where instead of comparing a variable to a constant which can be collected at the start, the program compares two variables. The exact values of the variables are unknown before the execution occurs, making the static seeding technique inapplicable. However, what can be done instead is that during the evaluation step of the algorithm where we execute the test case to measure its performance, we can also monitor and record the values of variables. We can add those values to a separate dynamic pool of data upon which the algorithm can draw upon during the next iteration of the crossover and mutation operations, potentially speeding up the process of finding the correct values to satisfy the conditions.

Type seeding attempts to address the problem where the function under test accepts a parameter of type `Object` which is the superclass of all objects in Java. This situation means that the search algorithm does not have enough information regarding what object to pass to the function. Instead, it will waste iterations on finding the object which will allow a test to execute as much of the code under test as possible. However, what we could do is inspect the function which accepts the `Object` type for presence of casting or `instanceof` operations. Those operations could help us understand what specific object the function is actually expecting even if its signature expects `Object` type objects. Nonetheless, if the expected object is complex, i.e. creation of this object requires passing specific parameters or making certain function calls, then type seeding is not as effective. This is because the algorithm also needs to spend time to find a way to construct that object correctly.

To address the issue of complex object creation, we could incorporate the knowledge captured in previous tests by supplying them as seeds. For example, consider that we have previously generated tests for the program under test or previously human written tests. Those tests could contain useful information which would otherwise take time and iterations for an evolutionary algorithm to discover and correctly incorporate into its solutions. Things like complex object creations where, to test a function, we need to pass a certain object to it which we first need to create through a factory or other objects. If we already have test cases which show examples of how to create those objects, we can attempt to carve the list of statements necessary for creating this object. The algorithm will attempt to reuse this list of statements to create this object rather than doing it from scratch. This strategy is known as carving. Another strategy known as cloning, focuses on the idea of copying the exact statements from previously available tests into newly generated ones. Both strategies are interesting and were shown to be beneficial to the search [46].

2.3 Large Language Models

Large Language Models (LLMs) have seen a growth in popularity and usage. Researchers also became interested in what LLMs can do and how to best work with them. This lead to the establishment of a new research field known as "Prompt Engineering" and production of many papers involving LLMs. The Evolutionary Computation community is no exception to this trend as it has quickly produced many papers exploring the possibilities of incorporating LLMs into evolutionary algorithms.

2.3.1 LLM Prompt Engineering

ChatGPT's main and only user interface was text-based input, a prompt, from the user to which the model would respond by generating a text-based response. Being the only means of interaction with the model, people started to wonder how this input could be constructed in such a way that allows LLM's output to be the most useful and valuable. This question became even more pressing as companies began charging users money depending on the size of their input and the models' output. This prompt engineering research led to a discovery of interesting and more effective ways of constructing an LLM prompt. For example, general prompts often lead to more general responses due to the lack of specificity; hence a more specific prompt can yield a more precise response [15] saving the user's money along with time spent reading the response. Additionally, we can add instructions on how to format the LLM's reply to our prompt thereby potentially reducing the amount of post-processing needed to extract and use the desired information [33]. As an example, consider asking the LLM to put any code snippets between backticks which are usually used to define code blocks in Markdown-based systems. By successfully placing all the code between backticks in the response, we can quickly locate those backticks and extract the code out of them. Without those backticks acting as identifiers for the code location, we would not have a clear indication of when the code begins and ends, leading to less efficient and more bug-prone parsing.

2.3.2 Evolutionary Computation & LLMs

The advances of large language models and their availability sparked the interest of the evolutionary computation community to explore if and how LLMs can be combined with evolutionary computation and whether it would be beneficial. A recent survey [56] by Wu et al. identified that there are two ways in which LLMs could be combined with EAs, LLM-enhanced EAs and EA-enhanced LLMs. Wu et al. further proposes a roadmap for the hybrid future of the two approaches, stating that the potential of LLM-enhanced EAs lies in the ability of an LLM to generate novel solutions and LLM's ability to generate optimization algorithms. As for the EA-enhanced LLM approaches, [56] mentions possibilities of EA-based LLM architecture search and prompt engineering. Another work [13] by Cai et al. explored the possible improvements of evolutionary computation via LLMs, most of which would belong to the category of LLM-enhanced EAs defined by Wu et al.. The improvements suggested by Cai et al. can be divided into three main categories LLM-

driven EA algorithms, LLM-based improvement of populations and individuals, and other improvements.

The category of LLM-driven EA algorithms includes suggestions such as LLM-assisted Evolutionary Strategy Selection and Evolutionary Operators via LLMs.

LLM-assisted Evolutionary Strategy Selection suggests using the LLM's knowledge to decrease the dependency on researcher's expertise for selecting an evolutionary strategy. An example of such an approach is work [30] by Liu et al. which introduced an LLM-Driven EA (LMEA) for solving combinatorial optimization problems such as the Travelling Salesman Problem (TSP). The LMEA algorithm essentially leaves most of the evolutionary algorithm work to the LLM and instead focuses on guiding LLM through the EA process. In each iteration of the search, LLM is instructed by LMEA to select present solutions and generate new solutions by performing crossover and mutation operation. Afterwards, the LLM evaluates the newly obtained individuals and chooses to incorporate them into the next iteration of the search. Experimental results showed promising results of handling instances of TSP in comparison to traditional methods.

Evolutionary operators via LLMs could take on three different forms, one where LLM is the evolutionary operator itself, another where LLM guides the genetic operators and the third form where LLM generates new or modified operators. The suggested reason by Cai et al. for using LLMs in one form or another in evolutionary operators is that it could "break the fixed mindset" of an EA caused by explicit and usually fixed mathematical definitions for operators. Examples of those approaches include but are not limited to works [11, 26, 29, 36] of Brahmachary et al., Hemberg et al., Liu et al. and Morris et al..

The category of LLM-based improvement of populations and individuals includes points such as strengthening of population design and assistance in handling complex high-dimensional data. The idea behind those two suggestions is that LLMs could provide a better starting point for the initial population by providing better individuals and refining population structures.

Lastly, Cai et al. suggests other possible improvements for Evolutionary Computation (EC) methods such as multimodal LLM expansion, adaptation to dynamic environment, and interactive EC. Cai et al. imagines how using multimodal LLMs that allow combining different types of inputs such as text, images, and sound could greatly expand the applicability of evolutionary computation. The interactive chat-based nature of LLMs could improve human interaction with EC algorithms enabling optimization through human input, and LLMs' extensive knowledge could make EC more adaptable to changes and challenges. Despite many suggested possibilities, Cai et al. recognizes the limitations of LLMs when facing issues with increased complexity and high dependency on prompt design and contextual constrains.

Chapter 3

Related Work

The previous chapter, Chapter 2, should have given you the necessary background to understand the key concepts of this study. This chapter will inform you of the related work surrounding the topic of LLM-based seeding for test case generation. In particular, we will begin with a result overview of studies comparing LLM generated test cases with the various existing tools. After comparing the LLM tests to other tests, we will move onto the hybrid approaches, which we split into sections: LLM-Based Tool Test Case Generation and Tool-Based LLM Test Case Generation. The distinction lies in whether the article focuses on improving LLM-generated tests by introducing additional tooling, or whether the article focuses on improving tool-generated tests through LLM utilization. Lastly, having introduced all those papers, we will point out the research gap which those papers do not close.

3.1 LLM vs Test Case Generation Tools Comparisons

One of the related research directions undertaken by various researchers (Abdullin et al., Ouedraogo et al., Schäfer et al., Tang et al., [58]) is comparing the performance and quality of tests generated by LLM's to those of classic tools like EvoSuite. Their work [3, 39, 49, 52, 58], provides us with the first data to guide our expectations with regard to LLM's generation abilities in the area of testing.

TestWars [3] paper by Abdullin et al. is the most recent comparison paper, out of the ones listed here. It compares different test generation tools to LLM's test generation ability. In it Abdullin et al. attempts to address the shortcomings of other papers by expanding the comparison to a symbolic execution-based tool Kex and using the GitBug data set instead of Defects4J because it is newer and therefore less likely to have been used in model training, which would lead to data leakage issues. Additionally, the paper computes several code feature metrics which impact test generation performance, such as number of dependencies, condition type, comments, and Java docs to identify the strengths and weaknesses of the tested tools. The experimentation is done with EvoSuite and Kex as traditional tools as well as four LLM models, namely ChatGPT-4, ChatGPT-4o, Llama Medium and Code Llama 70b. The results of the paper showed that LLM-based test generation falls behind traditional

methods in terms of coverage, since only the ChatGPT-4o model achieved a compilation rate greater than 7% and line coverage greater than 5%. The best metric of ChatGPT-4o was the median mutation score, where it achieved the highest value, 6.32% out of all competitors.

Other papers such as [39, 49, 52, 58] have focused primarily on evaluating the pure LLM test generation capability in contrast to other models or existing tools such as EvoSuite. For example, [39] conducted a study on four different LLM models (GPT-3.5, GPT-4, Mistral 7B, Mixtral 8x7B) in which they tried five different prompt engineering techniques (Zero-shot learning, Few-shot learning, Chain-of-Thought, Tree-of-Thoughts, Guided Tree-of-Thoughts). They compared their results with each other and also with EvoSuite as the state-of-the-art algorithm. Their findings were that prompt engineering can greatly impact the performance of LLMs, but EvoSuite generally continued to outperform LLMs. Another paper [49] by Schäfer et al. has investigated the ability of LLMs to generate tests for JavaScript programs. Additionally, they implemented a re-prompting mechanism in TestPilot for the LLM to try and fix the broken test cases it generates if possible. A study [52] by Tang et al. focused specifically on comparing ChatGPT's test generation capabilities with the EvoSuite's w.r.t. such as correctness, readability, code coverage, and bug detection capabilities. Furthermore, there was a study [58] by Yang et al. that performed similar comparisons but focused on the performance of closed-source models such as ChatGPT compared to open-source ones such as Code Llama.

3.2 LLM-Based Tool Test Case Generation

There have been several works which decided to make the LLM the core unit driving the test case generation process and supply it with additional pre- or post-processing steps and tools to improve the LLM performance. For example, Chen et al., developed a ChatUnitTest Core [14] framework which consists of five key steps, the preparation stage featuring the unit under test parsing and analysis, then followed by prompt construction where a chain-of-thought prompt technique is used. Once all preprocessing has been completed, the framework moves to the generation phase, where it extracts the tests and attempts to validate their correctness. If the tests are not functional, then it tries to fix them using a set of predefined rules, and if that fails the LLM is prompted again with details about the error and instructions to fix it. They evaluated their approach on a small set of projects w.r.t. line coverage in comparison to TestSpark and EvoSuite. The results of their approach were better than those of EvoSuite and TestSpark; however, some questions can be raised about the selected benchmark projects and the lack of statistical data.

Another paper [16] by Dakhel et al. devised an approach titled MuTAP the aim of which was to generate test cases with LLMs but then improve their effectiveness in revealing bugs by using mutation testing. Precisely, upon generating a test case, it was evaluated w.r.t. mutation score metric and if some mutants survived, the LLM would be prompted again with the example of a mutant and asked to improve the test case. The study was implemented for Python language and was evaluated in comparison with Pynguin. Their findings are that LLMs require specific post-processing, without which its effectiveness is not as good, and they would struggle to test corner cases or specific types of bugs.

Pizzorno and Berger have created an approach titled “CoverUp” [45] that also attempted to improve the test case generation performance of LLMs with regard to Python programmes. In their approach, they decided to first identify the code segments or units under tests which required more testing than what was already done in the existing test suites. Having identified the specific parts of improvement, they prompt the LLM to generate those test cases. However, in addition to providing the prompt, they also provide a tool function to the LLM to request more context if necessary. If LLM generates a new test that still lacks coverage or fails, the LLM is prompted again to fix the issue. The existing test suite is extended with successful test cases. Their evaluation shows improved performance compared to that of the CODAMOSA and MuTap approaches.

The work [47] by Ryan et al. tries to improve the generation of LLM-based test cases by addressing the approach of using fixed prompting strategies. In their approach, SymPrompt deconstructs the process of test suite generation into a multi-stage sequence. Then a specific prompt for each of the deconstructed sequences aligns with the execution paths of the unit under test. Once again, their work targets the Python-based project and showed improvement over their own baseline; however, there was no comparison with other tools like Pynguin.

A recent paper [51] by Straubinger et al. instructed the LLM to use the concept of “Scientific Debugging” to better understand how a generated unit test should be changed to kill the specified mutant. They compared the approach to three baselines, namely Pynguin, directly asking LLM to generate the test case without repeated querying in case of a failure, and asking LLM with repeated querying in case of a failure. Their findings show that despite the higher computational costs of using “Scientific Debugging”, this approach consistently outperforms Pynguin w.r.t. fault detection and coverage. Furthermore, they note the importance of iterative test case refinement to achieve higher-quality tests.

Lastly, the work [54] by Wang et al. has noted the pattern of other approaches to provide a complete unit under test to the LLM without providing any assistance in the input analysis. They hypothesized that this makes it more challenging for the LLM to infer test inputs to cover all conditions. Hence, resulting in a generated test suite with greater missing line and branch coverage. Their approach, HITS, tackles this problem by decomposing the methods into slices and querying the LLM to generate tests on those individual slices rather than the whole piece. Their approach is designed for a Java-based project and, according to their results, outperformed ChatUniTest and EvoSuite.

3.3 Tool-Based LLM Test Case Generation

Despite the relative recentness of using LLMs, in the evolutionary algorithm (EA), there were already several attempts at combining the two approaches in the unit test case generation field. The first approach, CODAMOSA [28], by Lemieux et al. attempted to escape the coverage plateaus encountered during the search process by querying an LLM. The main idea behind this method was that once the algorithm has noticed that it stalled, that is, there was no improvement for several iterations, it would send a prompt to an LLM, Codex, asking it to provide an example of a test case for an under-covered function. This approach

was implemented in Pynguin, a unit test case generation tool for Python programmes, and was evaluated on 486 benchmark modules over 27 projects assembled from datasets that were used in the evaluation of Pynguin [32] and BugsInPy [55]. The results of this study showed improved performance for LLM configurations, and a building block for another paper exploring the integration of LLMs into test case generation algorithms.

A paper [57] by Xiao et al. investigated how the LLM intervention could be applied in different parts of the EA. In this paper, an LLM intervention is applied in the three different stages of the search process, namely the initial stage, the test generation period, and the test coverage plateaus. It has mostly followed the same approach as the CODAMOSA [28] paper, for example, by using the same benchmark dataset and the same approach to adopting Code Llama responses in Pynguin format. Their results showed that the LLM intervention has a positive impact on the search process in any of the three investigation stages; however, there must be a reasonable frequency of the LLM intervention in each stage.

Another paper [59] involving Pynguin was written by Yang et al. which focused on the use of LLMs to address the problem of untyped code in Python. Python being a dynamically typed language can be more challenging to generate test cases for because of the missing information regarding the types necessary for the unit under test. The methodology proposed in [59] suggests a method for annotating parameter types in Python through an LLM in hopes of reducing the search space of the EA. Furthermore, they also experiment with a new mutation strategy involving Chain-of-Thought prompting. Lastly, they also employ an LLM for test repair in case the generated test suites contain errors, thereby increasing the executability of the generated tests. They did not use exactly the same data set as the other two studies; however, they did take some projects from the Pynguin [32] and added more from HumanEval[1] benchmark evaluation. Their results showed improvements in 16% of the modules.

Lastly, to our knowledge, the only paper which attempted to integrate LLM functionality into EvoSuite, a unit test case generation tool for Java projects, is [40] by Ouédraogo et al.. This work attempted to address the problem of generating relevant inputs for the units under test. In an attempt to generate more relevant input, they proposed a method of extracting the relevant input from various bug reports and using LLMs to help them do that. According to their results, their approach, BRMiner, is able to extract more and more relevant inputs, which when supplied to EvoSuite lead to an increased code coverage of the generated tests. Their experiment evaluation was conducted on the Defects4J benchmark, which can be considered a classic benchmark, however, is also more susceptible to being used in LLM training leading to potential data leakage issues.

3.4 Research Gap

Our research in various related work showed that there were many studies investigating the ability of an LLM to generate unit tests (Section 3.1), as well as several studies that tried to change or supplement the LLM with more information to get better tests (Section 3.2). However, there have only been a few attempts at integrating LLMs into existing evolutionary

[1] https://github.com/openai/human-eval

algorithm-based unit test case generation tools, especially EvoSuite, which works with Java projects (Section 3.3). Therefore, it would be interesting and valuable for the research community to explore how information from the LLMs can be used in combination with EvoSuite.

Chapter 4

LLM-Based Seeding for Test Case Generation

The main idea was to see how LLM's code generation ability could be useful at the initialization stage of the genetic algorithm. As discussed in the previous chapter, the initialization stage is the one where we create the initial population of the algorithm. It plays a crucial role because it sets the starting point for the search, which could be good or not. The starting point could be a local optimum rather than a global one, or it might not even be good at all.

We should also recall what it means for our starting population to be good. For that, remember that our population consists of individual test cases where the quality of a test case is determined by the custom fitness function measuring how close it comes to covering a certain goal, like line or branch. For a test case to be successful, it usually has to identify the right methods to call to reach the given target (line/branch) and the suitable input parameters or other factors influencing this particular targeted code. (arrange + act)

A random starting point will likely struggle with both of those parts because there are no guarantees on what kind of statements are inserted into the test case. It could take time before it inserts the correct method, and even longer before it finds or creates suitable inputs for the method which would lead to the necessary target.

Previous attempts at solving this problem included things like extracting information through already generated tests, e.g. by carving or cloning. However, this approach relies on the existence of those prior tests, which might not always be the case. This is where LLM's code generation abilities based on input could come in useful. We can prompt the LLM to create test cases and then supply them as a seed, and this is what we do on the high level. However, there are several details which are worth considering.

In essence, there are two parts, the LLM part and EvoSuite part. On the high level what we do is download the benchmark project which contains classes under test, run TestSpark for LLM-based test case generation on the downloaded project through a command line and then supply the generated tests into EvoSuite as a seed for carving or cloning strategies.

4.1 LLM-Based Test Case Generation Through TestSpark

The first part of the approach involves prompting the LLM to generate the test cases. There are different ways through which we could interact with an LLM, each providing a different set of benefits and drawbacks. For our purposes and constraints we chose to use TestSpark, an IntelliJ plugin, as a mean to interacting with an LLM because of its support for multiple LLM models and powerful code analysis capabilities provided by the headless IntelliJ IDEA. Headless IntelliJ allows us to run it on the server as a part of a pipeline.

The first advantage of using TestSpark instead of a custom implementation to interact with an LLM is the availability of different LLM models through different LLM platforms. Currently, TestSpark offers LLM platforms such as OpenAI, HuggingFace, Google AI, and JetBrains' AI Assistant platform. This variety of LLM providers gives us a bigger collection of LLM models on which we would like to run our experiments.

The second and more powerful advantage of TestSpark is its inherited capabilities of code analysis and inspection of IntelliJ IDEA, which could be useful for prompt creation. As discussed in the background chapter, prompt plays a crucial role in LLM's ability to generate a good output. Ideally, the prompt is detailed as to provide all the necessary information to the LLM, but also specific enough to prevent the LLM's response from going off-topic. Additionally, the size of the prompt cannot exceed a certain context limit, different for each model. TestSpark's code analysis features enable it to extract various code-related details from the various parts of the project which could be useful for the prompt. But it also features a check for prompt size limitations and a feedback mechanism checking compilation of LLM generated test cases.

```
Generate unit tests in Java for $NAME to achieve 100% line coverage for this class.
Dont use @Before and @After test methods. Make tests as atomic as possible.
All tests should be for JUnit 4. In case of mocking, use Mockito.
But, do not use mocking for all tests.
Name all methods according to the template
                    [MethodUnderTest][Scenario]Test, and use only English letters.
The source code of class under test is as follows:
$CODE
$METHODS
$POLYMORPHISM
```

Figure 4.1: Default LLM prompt in TestSpark.

TestSpark's default prompt listed in Figure 4.1 includes in it details like the actual code of the class under test `$CODE`, the method signatures available to it during test generation (`$METHODS`), and polymorphic relations (`$POLYMORPHISM`). It can include information such as about polymorphic relations in its prompts thanks to the inherited code processing, analysing, and extraction features of IntelliJ. However, the final size of a prompt like this is highly dependent on the code under test and its size. In some cases, the source code part of the prompt itself can easily overflow the context limits. To address this issue, TestSpark checks the final size of the prompt against the context limit size and, if necessary, reduces the prompt size by iteratively removing details. For example, it will first try to remove

the `$POLYMORPHISM` information; then should this not be enough, it will also remove the method signatures (`$METHODS` resulting in only source code (`$CODE`) being included in the prompt. If the source code itself is too large for the given model, then the TestSpark stops the test case generation process with the according error message, otherwise it proceeds to generate tests.

Having met the prompt's size constraints of an LLM model, TestSpark sends the generated prompt and receives the response. However, successful receiving of the response does not equal to a functional LLM test case or test suite. LLM's response could be anything ranging from no code at all to a complete functional test case. Parsing and validating this response is another advantage of using TestSpark as an engine for LLM-based test case generation instead of having to implement it ourselves and dealing with various edge cases. Once the response is received, it is parsed and decomposed into single unit tests rather than a whole test suite. Individual tests are saved into separate files which attempt to preserve all the necessary dependencies and imports for the test case to function, but it cannot guarantee the presence or existence of LLM hallucinated parts. To check for LLM hallucination, or more precisely for the functionality of the generated tests, TestSpark attempts to compile each file. If the compilation is successful, the test case is kept, but should it encounter an error, then TestSpark will attempt to fix it by including the compilation error message in a new prompt to the LLM. This feedback loop could proceed indefinitely because there are no guarantees that LLM can arrive at a functional solution, therefore, the user can choose themselves how many iterations they can afford.

The process described above attempts to generate an entire test suite from a given code snippet; a potential drawback of such an approach is that size also limits the LLM response, similar to the input. The LLM could run out of token to complete its response, leading to an incomplete answer which is hard to parse and is simply discarded. A possible solution to this problem would be to attempt a more precise LLM prompting approach. Instead of requesting an entire test suite given all the source code, we could specifically ask for certain details such as input parameters or objects to cover a certain target. However, the difficulty with such an approach lies in the fact that it is EvoSuite which is responsible for identifying targets and integrating LLM prompting capabilities, similar to TestSpark's, into EvoSuite is rather challenging given the time constraints. The details of the encountered technical challenges are detailed in Section **??**.

4.2 LLM-based EvoSuite Seeding

Having LLM generated the tests using TestSpark, we now need to provide them as a seed to the EvoSuite. This should be relatively easy because other researchers, namely, Rojas et al. have previously implemented such functionality in the EvoSuite when investigating [46] the carving and cloning strategies. However, the challenge here lies in the fact that the work of Rojas et al. was done before, Panichella et al. reformulated the test case generation into a multi-objective problem.

EvoSuite is a mature tool as it started in 2011 [19] and throughout the years it became highly customizable thanks to the efforts of many researches who have developed various

improvements to try out new strategies and approaches. However, this scattered among many researchers development process did not end up in an easy-to-use tool where all options are compatible with each other. This is particularly true when Panichella et al. began implementing a multi-objective approach in EvoSuite. In their implementation, Panichella et al. relied on a random way to create the offspring, and therefore the seeding-based approach of Rojas et al. involving previously generated tests was not supported. Hence we had to slightly modify EvoSuite's codebase to support the seeding approach in the DynaMOSA algorithm. After implementing our changes and testing them, we found a couple of inconsistencies which helped us resolve some issues in our own modifications but also find a bug in EvoSuite. Specifically, the problem was in target duplication among the uncovered and covered parts of the archive. When adding a new target of type "exception" to the uncovered archive, its presence was not checked in the covered archive. This leads to target duplication where if the target was already covered and placed into the covered archive, it would be added again into the uncovered. After fixing this issue, the algorithm behaviour became more consistent.

Having modified EvoSuite to enable carving and cloning strategies to the best test case generation algorithm, we were now able to use EvoSuite CLI to specify the project under test along with other properties and supply compiled LLM tests. The carving and cloning strategies can be chosen by specifying the according flags and parameters.

Chapter 5

Empirical Evaluation

5.1 Research Questions

As discussed in the previous chapters, the starting point of the genetic algorithm is the initial population, which has a great impact on the direction the search goes and the discovered solutions. Previous work attempted to use prior knowledge, such as already generated test [46], to improve the starting point of the search algorithm. Although the results seemed promising, one of the encountered issues with this approach was the availability of prior knowledge in the form of test cases. This is where our investigation comes in, as it aims to explore the possibility of using an LLM to generate the missing test and use them as a seed for EvoSuite. Our concrete research question for this study is:

RQ: What is the impact of ChatGPT-4o generated tests when supplied as a seed for carving and cloning strategy on EvoSuite's test case generation?

Although it would be interesting to explore the capabilities of different models, previous work such as [3] showed that due to context size limitations, some models are struggling to even initiate the generation of those test cases. Thus, we will rely on OpenAI's ChatGPT-4o as a starting model for our study because it was shown to have the highest test case compilation rate in [3].

The impact of the seed in the form of LLM-generated test cases can be measured through the performance of the newly generated test case via the classic criteria such as line coverage, branch coverage, and mutation score. However, to answer the question of "what is the impact" and determine which strategy, cloning or carving, benefited more, we would need to compare the results. Therefore, to answer this question, it is important to run four different configurations: baseline, carving, cloning and combination of carving and cloning referred to as "combined". The baseline configuration serves as the control group with the default EvoSuite behaviour using its best performing algorithm. While carving, cloning, and combined configurations are the experimental groups where we supply the same LLM-generated tests as a seed but turn on different seeding strategies. Results of those four configurations can be compared and contrasted with one another, leading to an understanding if the treatment groups improve over the baseline, therefore signifying the usefulness of LLM generated test cases or not. Additionally, those results will help us see

if one of the strategies performs better than the other, giving us the insight into what exactly was useful about the LLM generated test cases.

5.2 Benchmark

The benchmark we chose for conducting this experiment is the same as in paper by Abdullin et al., namely GitBug. Other related work has mostly used different dataset such as Defects4J or EvoSuite's SF110 benchmarks. However, similarly to Abdullin et al. we believe that using a newer dataset such as GitBug will make the experiment less susceptible to the data leakage issue.

The GitBug Java data set is a collection of 199 bugs from 55 open-source projects which were assembled to create a reproducible benchmark of recent Java bugs. For our experimentation the benchmark is reduced to 136 projects similarly as in [3] the case because of compatibility issues with the EvoSuite. Specifically, EvoSuite only works on projects which are developed on Java 11. While some projects in the GitBug data set were written in Java versions higher than 11. The dataset presents a range of projects with the classes under test (CUT) ranging in source lines of code (SLOC) from 25 to 2,500 and with the average cyclomatic complexity of approximately 66. However, it should be noted that there is a project imbalance caused by the presence of 29 `jsoup` related CUTs and 70 CUTs related to the `traccar` project out of 136 total CUTs.

5.3 Parameters

In our case we need to set parameters for TestSpark and EvoSuite. Additionally we are using Abdullin et al.'s TGA-Pipeline developed for [3] to run experiments on TestSpark and EvoSuite.

5.3.1 TestSpark-LLM

The main parameter in the TestSpark important for our experiment is the prompt for the LLM. In Chapter 4 we discussed in more detail what kind of prompt we are using and why. Please see that section if you wish to. But in short it is the default TestSpark prompt which includes things like source code of a CUT, method signatures, and polymorphic relationship, but shrinks in size if it becomes too big for LLM's context window. Additionally, we use default TestSpark settings for the feedback mechanism. Precisely `The depth of input parameters used in class under tests` is set to 2, `Maximum polymorphism depth` is also 2, while `Maximum number of requests to LLM` is 3. For completeness it should be mentioned that LLM capabilities of TestSpark require an authentication token specific to the chosen AI platform, which we also provide during our experiments. In our case this is JetBrains' AI Assistant platform in which we use ChatGPT-4o.

5.3.2 EvoSuite

EvoSuite also has parameters which we need to set. Similarly to TestSpark, most parameters are set at default values; however, certain adjustments need to be made for the specific configurations. For the initial population we use the commonly used value of 50 individuals. While the search budget is set at the standard 120 seconds as in many competitions [24, 25, 35, 43]. We use the default and best performing, according to competitions, genetic algorithm DynaMOSA as the main search strategy. The goal criteria is the default set of target consisting of line, branch, exception, weak mutation, output, method, method exception, and branch objectives. Those parameters make up the baseline configuration and do not change for the carving and cloning configurations.

For the carving and cloning configurations, the important change we need to make is to add the canonical names of classes or test cases that we want to use as our seed into the `selected_junit` CLI parameter. Furthermore, each configuration, carving and cloning, brings its own set of adjustable parameters.

For example, the carving configuration has a boolean flag `carve_object_pool`, which needs to be set to `true`, and `p_object_pool` probability which is responsible for determining how often we draw objects from the carved object pool rather than the other pools. While the cloning configuration has additional parameters such as `seed_clone` and `seed_mutations`. Former, `seed_clone`, is responsible for determining the probability of how often we insert statements from the supplied LLM generated test. While the latter, `seed_mutations`, sets the number of mutations we make to those drawn statements. In our case we use optimal probabilities suggested by [46] EvoSuite numbers for all those additional adjustable parameters, concretely `p_object_pool = 0.9`, `seed_clone = 0.9`, and `seed_mutations = 8`.

Note that to ensure that cloning is not used during the carving configuration and vice versa, we specifically set the adjustable parameters of those strategies to 0. In other words, carving configuration has parameter values like `carve_object_pool=true`, `p_object_pool=0.9`, `seed_clone=0.0`, and `seed_mutations=0`. While cloning configuration has parameter values such as `carve_object_pool=false`, `p_object_pool=0.0`, `seed_clone=0.9`, and `seed_mutations=8`.

5.4 Experimental Protocol

Due to the stochastic nature of EvoSuite's DynaMOSA search algorithm and LLM's test case generation it is important to perform several iterations per each configuration, as suggested by [9] and is done in previous related work. Therefore for each one of the four configurations, baseline, carving, cloning, and combined we run the experiments ten times. Precisely, we first generate a set of LLM tests through TestSpark where we have 136 benchmark projects times 10 iterations to account for LLM randomness. We do not need to do it three times because the base line configuration does not require LLM tests. While carving and cloning configurations should use the same LLM tests as seeds to make the comparison between them fairer. Once we have the LLM generated tests we can move to generating tests with EvoSuite. We run the baseline configuration 136×10 times. And the same goes

for carving and cloning configuration, for all of which we supply the same LLM generated test. Note that a small time optimisation we can do is reuse the already generated ChatGPT-4o tests from [3] replication package[1]. To further speed up the computation we run the experiments on the same server with 128 cores and parallise up to 10 instances of docker containers (total is 20 containers because we have base and tool). We run TGA-Pipeline's analysis to compute line, branch coverage and mutation score.

5.5 Threats to Validity & Reproducibility

5.5.1 Construct Validity

Threats to construct validity refer to the study's ability to measure what it claims to measure. In our case, we are interested in finding out if LLM-generated tests are suitable for carving and cloning strategies and what impact they have as a seed. We measure the impact through metrics such as line coverage, branch coverage, and mutation score given the same time-based search budget. This approach is widely adopted in the field of search-based software testing, as it provides a reasonable estimation of the effectiveness and efficiency of the tried test case generation technique.

5.5.2 Internal Validity

Internal validity considers factors arising from our implementation that could influence our results. In our case, we are working on three different open source projects, TestSpark[2], EvoSuite[3], TGA-Pipeline[4], as well as our own repository containing various utility scripts[5], all available on GitHub. In our study, we have particularly added new functionality to EvoSuite's fork on this branch and TGA-Pipeline fork on this branch. Additionally, we provide a replication package[6] that contains the obtained results. Due to the nature of software development, each of the projects is susceptible to issues and bugs. The projects cannot guarantee a bug-free implementation because tests can only show the presence of bugs, not their absence. Despite our best attempts at checking and verifying the functionality of the code, we cannot give guarantees but are happy to share our code with others, hence it can be found on GitHub, for future examination and improvement.

5.5.3 External Validity

External validity attempts to account for factors outside our implementation. One of those factors that specifically stems from the LLMs is the issue of data leakage. LLMs are trained on a large corpus of data that covers a wide range of things, and the specific contents of these training data are rarely disclosed to the public. The LLM performs better when asked

[1] https://zenodo.org/records/13862019
[2] https://github.com/JetBrains-Research/TestSpark
[3] https://github.com/ciselab/evosuite/tree/SergeyDatskiv/thunderdome-MOSuiteStrategy-CarvedTests
[4] https://github.com/SergeyDatskiv/tga-pipeline/tree/SergeyDatskiv/development
[5] https://github.com/SergeyDatskiv/TUDelft-MScThesis-PromptSeedGenerate
[6] Replication Package is stored on Zenodo under the following doi:10.5281/zenodo.15698634

questions about the data to which it has previously been exposed rather than new information. Since we do not know exactly what data were used to train the model, we cannot be sure that the results of our experiments will be generalized to other datasets or models. GitBug benchmark is a relatively new benchmark, meaning that the likelihood of it being used in training is smaller compared to classic data sets such as Defects4J and SF110. However, despite the recency of GitBug, it should be noted that the individual projects that make up this dataset are not necessarily new. They, too, could have been used in the training of LLM models as a standalone project or as a part of some other data set.

In addition to the problem of data leakage, there is also the possibility that the benchmark is not diverse enough. Lack of diversity in the data set can lead to poor generalisation of the performance. Projects that are not covered by GitBug may have different characteristics, and test case generation can perform significantly differently for the better or worse. Although this concern is true for virtually all benchmarks, it is particularly true for GitBug, which out of 136 test units has a project imbalance, with 29 units being from the `jsoup` project and 70 from the `traccar`.

We also run our experiments on the same hardware to prevent any differences from making an impact.

5.5.4 Conclusion Validity

The validity of the conclusion relates to the certainty and reliability of the conclusion we draw from our results. In our particular case, one of the biggest possible issues is the stochastic nature of LLMs and EvoSuite's genetic algorithm. To reduce these concerns, we repeat the experiments in EvoSuite ten times per configuration per test unit ($136 \times 10 \times 3$). In terms of generating LLM test cases through TestSpark, we reuse existing test cases from the [3] replication package, but they were also repeated ten times per test unit (136×10). Furthermore, the tests described by Abdullin et al. in [3] were generated in isolation, that is, using different prompting sessions, to prevent the LLM from learning from previous input. Regarding our conclusions, we draw them based on well-established metrics such as line coverage, branch coverage, and mutation score and compute well-established statistical values for them such as Mann-Whitney U tests [37] to determine statistical significance and Vargha-Delaney $\hat{A}_{12}$ [53] statistic for effect size.

Chapter 6

Results & Discussion

This study aims to investigate the following question.

RQ: What is the impact of ChatGPT-4o generated tests when supplied as a seed for carving and cloning strategies on EvoSuite's test case generation?

To answer this question, we performed four different configurations, baseline, carving, cloning, and combination of carving and cloning, also known as "combined". The configurations follow our LLM-based seeding approach described in Chapter 4. In the following, we present the results and discuss them.

6.1 Results

6.1.1 Line Coverage Metric

Firstly, let us examine the impact of seeding through carving, cloning, and combined (carving and cloning together) strategies in comparison to the baseline on line coverage.

Table 6.1 shows the Mann-Whitney U Tests p-value and Vargha-Delaney $\hat{A}_{12}$ effect sizes for line coverage when comparing the baseline to different seeding configurations. The highlighted cells show a statistically significant difference from the baseline (p-value ¡ 0.05), where an $\hat{A}_{12}$ effect size score greater than 0.5 shows better performance for the corresponding seeding configuration, while a lower score indicates better baseline performance. From the table, we can see that there are only nine classes that are statistically significant. Of those nine classes, eight show a positive effect of the seeding configurations, while only one indicates a better performance for the baseline. In four cases (`jsoup-78aeac18c6`, `jsoup-eff15210b0` , `semver4j-48ffbfd1f6`, `traccar-ec2b7b64a8`) there is a benefit from the three configurations, carving, cloning, and combined. Meanwhile, the other four results are split into the three configurations with the following distributions: two for the combined configuration (`traccar-074dc016d2`, `word-wrap-930eb5e91a`), one for cloning (`crawler-commons-d8a6126365`), and one for carving configurations (`jsoup-4864af45af`). As for the result, that shows that the baseline performed better, that is, only when compared to the combined configuration.

The biggest takeaway from this table is that in about 81% of the cases, the line coverage does not seem to improve with any of the seeding strategies. Despite that, there are some

cases where seeding makes the difference for the better and worse. It seems that there is no single configuration that constantly performs better than others, since for each configuration there exist cases where at least one class improves only for that configuration. Additionally, there exist classes where any one of the seeding configurations will make a positive difference.

Benchmark	File	Baseline vs Combined		Baseline vs Carving		Baseline vs Cloning	
		p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
ConfigMe-7449db4901	CommentsConfiguration	1.000	0.500	1.000	0.500	1.000	0.500
ConfigMe-cab40d1c3c	PropertyListBuilder	0.408	0.400	0.204	0.350	0.083	0.300
aws-java-nio-spi-for-s3-890fdddf30	S3Path	0.786	0.535	0.073	0.715	0.159	0.670
aws-secretsmanager-jdbc-42dc301cc7	AWSSecretsManagerPostgreSQLDriver	1.000	0.500	1.000	0.500	1.000	0.500
crawler-commons-d8a6126365	SimpleRobotRulesParser	0.704	0.555	0.061	0.750	0.007	0.855
crowdin-api-client-java-334d414753	JacksonJsonTransformer	1.000	0.500	1.000	0.500	1.000	0.500
epubcheck-8575a6b1c4	EpubChecker	0.121	0.290	0.568	0.420	0.471	0.400
formatter-maven-plugin-a6994326aa	CssFormatter	1.000	0.500	1.000	0.500	1.000	0.500
java-solutions-7a73ea56d0	OnlineStockSpan	1.000	0.500	1.000	0.500	1.000	0.500
java-stellar-sdk-06641953c4	KeyPair	0.328	0.615	1.000	0.505	0.871	0.520
java-stellar-sdk-d6379e9615	Transaction	0.593	0.575	0.791	0.540	0.909	0.520
jsoup-13f7ef9241	DataNode	1.000	0.500	1.000	0.500	1.000	0.500
jsoup-23573ef31c	Document	0.102	0.320	0.081	0.285	0.926	0.515
jsoup-23ea77ef4b	W3CDom	0.562	0.420	0.938	0.515	0.088	0.725
jsoup-4864af45af	Safelist	0.061	0.740	0.015	0.815	0.606	0.430
jsoup-78aeac18c6	CharacterReader	0.001	0.900	0.001	0.900	0.004	0.850
jsoup-9170b1d17b	Attributes	0.006	0.140	0.429	0.400	0.108	0.295
jsoup-b129bc9e3b	HttpConnection	0.211	0.670	0.405	0.615	0.240	0.660
jsoup-c507588b5c	TextNode	1.000	0.500	1.000	0.500	0.368	0.450
jsoup-c61ce94f35	StructuralEvaluator	1.000	0.500	1.000	0.500	1.000	0.500
jsoup-e1880ad73e	UrlBuilder	1.000	0.505	0.670	0.460	0.626	0.455
jsoup-eff15210b0	HttpConnection	0.001	0.945	0.0	0.965	0.002	0.905
jsoup-f0eb6bd1cc	UrlBuilder	0.583	0.450	1.000	0.500	0.278	0.395
markedj-2881d5b547	Marked	1.000	0.500	1.000	0.500	1.000	0.500
nfe-ec5ddf7e73	MDFInfoModalRodoviarioVeiculoReboque	1.000	0.500	1.000	0.500	1.000	0.500
semver4j-48ffbfd1f6	XRangeProcessor	0.033	0.760	0.023	0.780	0.001	0.915
semver4j-bf853ab269	Semver	1.000	0.500	1.000	0.500	1.000	0.500
traccar-074dc016d2	MediaFilter	0.026	0.770	0.585	0.570	0.773	0.540
traccar-1a1126d2d3	HuaShengProtocolDecoder	1.000	0.500	1.000	0.500	1.000	0.500
traccar-28440b7726	RuptelaProtocolDecoder	1.000	0.500	1.000	0.500	1.000	0.500
traccar-2ac77554f7	T55ProtocolDecoder	0.756	0.460	0.691	0.455	0.894	0.480
traccar-2dd48fa51b	GatorProtocolEncoder	0.821	0.470	0.966	0.490	0.761	0.540
traccar-3ba077b000	TramigoFrameDecoder	0.063	0.740	0.195	0.670	0.289	0.640
traccar-532c414196	WatchFrameDecoder	0.467	0.405	0.562	0.425	0.784	0.540
traccar-596036dc33	Jt600FrameDecoder	0.466	0.600	0.377	0.620	0.516	0.410
traccar-6e5481ebb1	SuntechProtocolDecoder	1.000	0.500	1.000	0.500	1.000	0.500
traccar-7ade92a97f	LaipacProtocolDecoder	0.810	0.465	0.707	0.550	0.343	0.375
traccar-94fbc93f8b	GalileoFrameDecoder	0.275	0.640	0.155	0.685	0.668	0.560
traccar-9a427527da	Minifinder2ProtocolEncoder	0.368	0.550	1.000	0.500	1.000	0.500
traccar-a07e078645	Tk103ProtocolDecoder	0.452	0.600	0.514	0.585	0.269	0.645
traccar-a24d7d5d7a	FreematicsProtocolDecoder	0.463	0.595	0.655	0.560	0.592	0.570
traccar-ab801e8565	Tk103ProtocolDecoder	0.141	0.690	0.285	0.630	0.637	0.560
traccar-b3c6e22fc1	BufferUtil	1.000	0.500	1.000	0.500	1.000	0.500
traccar-b5d5ec4318	Gt06ProtocolDecoder	0.084	0.270	0.083	0.270	0.393	0.385
traccar-ec2b7b64a8	Parser	0.005	0.855	0.012	0.810	0.021	0.775
traccar-f1470e5670	WatchFrameDecoder	0.702	0.445	0.620	0.430	0.470	0.600
traccar-f73263da48	WialonProtocolDecoder	0.693	0.550	0.398	0.400	0.693	0.450
word-wrap-930eb5e91a	WordWrap	0.035	0.765	0.938	0.515	0.443	0.395

Table 6.1: Table showing the Mann-Whitney U Tests p-value and Vargha-Delaney $\hat{A}_{12}$ effect size when comparing baseline to different seeding configurations w.r.t. line coverage. Highlighted p-value indicates a statistically significant difference from the baseline (p-value $<$ 0.05), where a $\hat{A}_{12}$ effect size score greater than 0.5 shows better performance for the corresponding seeding configuration, while a lower score indicates better baseline performance.

6.1.2 Branch Coverage

Secondly, similar to line coverage analysis, let us examine the impact of seeding strategies compared to the baseline on branch coverage.

Table 6.2 presents an identical layout to Table 6.1 with the only difference being that the compared values correspond to branch coverage rather than line coverage. Similarly to the impact of line coverage, branch coverage does not appear to be influenced by any of the seeding strategies in 81% of the cases. Branch coverage seems more likely to be negatively impacted by seeding configurations, since four (`aws-java-nio-spi-for-s3-890fdddf30`, `jsoup-23573ef31c`, `jsoup-9170b1d17b`, `jsoup-c507588b5c`) out of nine show better performance with the baseline. The worse performance seems to occur across different configurations, three times in combined configuration (`aws-java-nio-spi-for-s3-890fdddf30`, `jsoup-23573ef31c`, `jsoup-9170b1d17b`) and once in carving configuration

(`aws-java-nio-spi-for-s3-890fdddf30`) and cloning (`jsoup-c507588b5c`). Regarding the remaining five positive improvements for the seeding configurations, we can see that in three cases (`jsoup-78aeac18c6`, `semver4j-48ffbfd1f6`, `traccar-ec2b7b64a8`) it is once again that all three configurations show improved performance. While the remaining two positive improvements go, one each for standalone carving (`jsoup-4864af45af`) and standalone cloning configurations (`crawler-commons-d8a6126365`).

Once again, the biggest takeaway seems to be that LLM-based seeding makes little significant difference apart from specific classes where it could benefit or disadvantage the branch coverage. Not a single configuration seems to clearly stand out as the most or least beneficial.

Benchmark	File	Baseline vs Combined		Baseline vs Carving		Baseline vs Cloning	
		p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
ConfigMe-7449db4901	CommentsConfiguration	1.000	0.500	1.000	0.500	1.000	0.500
ConfigMe-cab40d1c3c	PropertyListBuilder	0.408	0.400	0.204	0.350	0.083	0.300
aws-java-nio-spi-for-s3-890fdddf30	S3Path	0.036	0.250	0.021	0.220	0.670	0.540
aws-secretsmanager-jdbc-42dc301cc7	AWSSecretsManagerPostgreSQLDriver	1.000	0.500	1.000	0.500	1.000	0.500
crawler-commons-d8a6126365	SimpleRobotRulesParser	0.518	0.590	0.157	0.690	0.012	0.835
crowdin-api-client-java-334d414753	JacksonJsonTransformer	1.000	0.500	1.000	0.500	1.000	0.500
epubcheck-8575a6b1c4	EpubChecker	0.270	0.350	0.446	0.605	0.705	0.445
formatter-maven-plugin-a6994326aa	CssFormatter	1.000	0.500	1.000	0.500	1.000	0.500
java-solutions-7a73ea56d0	OnlineStockSpan	1.000	0.500	1.000	0.500	1.000	0.500
java-stellar-sdk-06641953c4	KeyPair	0.226	0.640	0.670	0.460	0.871	0.520
java-stellar-sdk-d6379e9615	Transaction	0.760	0.545	0.879	0.525	1.000	0.505
jsoup-13f7ef9241	DataNode	1.000	0.500	1.000	0.500	1.000	0.500
jsoup-23573ef31c	Document	0.047	0.250	0.052	0.250	0.865	0.475
jsoup-23ea77ef4b	W3CDom	0.937	0.515	0.841	0.470	0.272	0.645
jsoup-4864af45af	Safelist	0.062	0.750	0.011	0.840	0.789	0.460
jsoup-78aeac18c6	CharacterReader	0.005	0.860	0.002	0.910	0.004	0.880
jsoup-9170b1d17b	Attributes	0.005	0.130	0.814	0.465	0.174	0.320
jsoup-b129bc9e3b	HttpConnection	0.937	0.515	0.667	0.440	0.640	0.565
jsoup-c507588b5c	TextNode	0.183	0.665	0.259	0.645	0.017	0.195
jsoup-c61ce94f35	StructuralEvaluator	1.000	0.500	1.000	0.500	1.000	0.500
jsoup-e1880ad73e	UrlBuilder	0.211	0.370	0.386	0.415	0.386	0.415
jsoup-eff15210b0	HttpConnection	0.190	0.670	0.429	0.605	0.345	0.625
jsoup-f0eb6bd1cc	UrlBuilder	0.583	0.450	1.000	0.500	0.255	0.390
markedj-2881d5b547	Marked	1.000	0.500	1.000	0.500	1.000	0.500
nfe-ec5ddf7e73	MDFInfoModalRodoviarioVeiculoReboque	1.000	0.500	1.000	0.500	1.000	0.500
semver4j-48ffbfd1f6	XRangeProcessor	0.033	0.760	0.023	0.780	0.001	0.930
semver4j-bf853ab269	Semver	1.000	0.500	0.368	0.450	1.000	0.500
traccar-074dc016d2	MediaFilter	1.000	0.500	1.000	0.500	1.000	0.500
traccar-1a1126d2d3	HuaShengProtocolDecoder	1.000	0.500	1.000	0.500	1.000	0.500
traccar-28440b7726	RuptelaProtocolDecoder	1.000	0.500	1.000	0.500	1.000	0.500
traccar-2ac77554f7	T55ProtocolDecoder	0.520	0.420	0.307	0.380	0.701	0.450
traccar-2dd48fa51b	GatorProtocolEncoder	0.681	0.450	0.830	0.470	1.000	0.500
traccar-3ba077b000	TramigoFrameDecoder	0.063	0.740	0.195	0.670	0.221	0.660
traccar-532c414196	WatchFrameDecoder	0.491	0.410	0.562	0.425	0.784	0.540
traccar-596036dc33	Jt600FrameDecoder	0.376	0.620	0.317	0.635	0.444	0.395
traccar-6e5481ebb1	SuntechProtocolDecoder	1.000	0.500	1.000	0.500	1.000	0.500
traccar-7ade92a97f	LaipacProtocolDecoder	0.429	0.400	0.542	0.580	0.413	0.390
traccar-94fbc93f8b	GalileoFrameDecoder	0.275	0.640	0.155	0.685	0.528	0.585
traccar-9a427527da	Minifinder2ProtocolEncoder	0.368	0.550	1.000	0.500	1.000	0.500
traccar-a07e078645	Tk103ProtocolDecoder	0.452	0.600	0.660	0.560	0.269	0.645
traccar-a24d7d5d7a	FreematicsProtocolDecoder	0.301	0.600	0.301	0.600	1.000	0.500
traccar-ab801e8565	Tk103ProtocolDecoder	0.078	0.730	0.285	0.630	0.738	0.545
traccar-b3c6e22fc1	BufferUtil	1.000	0.500	1.000	0.500	1.000	0.500
traccar-b5d5ec4318	Gt06ProtocolDecoder	0.061	0.250	0.065	0.255	0.360	0.375
traccar-ec2b7b64a8	Parser	0.005	0.855	0.012	0.810	0.021	0.775
traccar-f1470e5670	WatchFrameDecoder	0.758	0.455	0.617	0.430	0.321	0.635
traccar-f73263da48	WialonProtocolDecoder	0.693	0.550	0.398	0.400	0.693	0.450
word-wrap-930eb5e91a	WordWrap	0.179	0.680	1.000	0.495	0.969	0.510

Table 6.2: Table showing the Mann-Whitney U Tests p-value and Vargha-Delaney $\hat{A}_{12}$ effect size when comparing baseline to different seeding configurations w.r.t. branch coverage. Highlighted p-value indicates a statistically significant difference from the baseline (p-value < 0.05), where a $\hat{A}_{12}$ effect size score greater than 0.5 shows better performance for the corresponding seeding configuration, while a lower score indicates better baseline performance.

6.1.3 Mutation Score

Following in line with the previous two sections 6.1.1 and 6.1.2, now we will look at the p-value and $\hat{A}_{12}$ values for the mutation score metric in Table 6.3.

Once again, there are only a handful of values with statistical significance. In total, there are 13 benchmark classes that have a p-value less than 0.05. Six of those 13 classes show that the baseline performed better than the seeding configurations. Interestingly enough, not a single one of those six classes corresponds to the standalone carving configuration. Instead, we have four classes (`aws-java-nio-spi-for-s3-890fdddf30`, `jsoup-23573ef31c`, `jsoup-9170b1d17b`, `traccar-b3c6e22fc1`) with better baseline performance for the combined configuration and the remaining two (`jsoup-13f7ef9241`, `jsoup-c507588b5c`) for the cloning configuration. Regarding the positive improvement due to seeding, we can see two classes (`jsoup-78aeac18c6`, `jsoup-b129bc9e3b`) with improvement in all three configurations. The remaining five classes have improvements in the following configurations: `jsoup-4864af45af` improved in carving configuration, `jsoup-b129bc9e3b` improved in combined and carving configurations, `semver4j-48ffbfd1f6` improved in combined and cloning configurations, `semver4j-bf853ab269` improved in cloning configuration, and `traccar-ec2b7b64a8` improved in combined and carving configurations. Overall, we can see once again that there is no obvious pattern in which configuration leads to which improvement. Sometimes any seeding strategy will help improve the mutation score metric, in other times it might be more specific to a particular class under test and could either improve or worsen the score. The one interesting observation is that the only configuration which seems to not have any downsides w.r.t. mutation score is standalone carving. Perhaps the reasons for this could be further investigated in the future; however, as it stands, no concrete conclusions can be drawn from the table as to why this happened. After all, this pattern could be very specific to the given set of projects.

Benchmark	File	Baseline vs Combined		Baseline vs Carving		Baseline vs Cloning	
		p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
ConfigMe-7449db4901	CommentsConfiguration	1.000	0.500	1.000	0.500	1.000	0.500
ConfigMe-cab40d1c3c	PropertyListBuilder	0.408	0.400	0.204	0.350	0.064	0.280
aws-java-nio-spi-for-s3-890fdddf30	S3Path	0.048	0.235	0.849	0.470	0.761	0.545
aws-secretsmanager-jdbc-42dc301cc7	AWSSecretsManagerPostgreSQLDriver	1.000	0.500	1.000	0.500	1.000	0.500
crawler-commons-d8a6126365	SimpleRobotRulesParser	1.000	0.500	1.000	0.500	1.000	0.500
crowdin-api-client-java-334d414753	JacksonJsonTransformer	0.077	0.350	0.167	0.400	0.368	0.450
epubcheck-8575a6b1c4	EpubChecker	0.285	0.355	0.157	0.310	0.298	0.360
formatter-maven-plugin-a6994326aa	CssFormatter	0.204	0.650	0.398	0.600	1.000	0.500
java-solutions-7a73ea56d0	OnlineStockSpan	1.000	0.500	1.000	0.500	1.000	0.500
java-stellar-sdk-06641953c4	KeyPair	0.307	0.620	1.000	0.500	1.000	0.505
java-stellar-sdk-d6379e9615	Transaction	0.789	0.540	0.970	0.510	0.970	0.510
jsoup-13f7ef9241	DataNode	0.137	0.320	0.398	0.400	0.032	0.250
jsoup-23573ef31c	Document	0.043	0.260	0.080	0.280	0.257	0.360
jsoup-23ea77ef4b	W3CDom	0.619	0.570	0.673	0.560	0.249	0.655
jsoup-4864af45af	Safelist	0.150	0.695	0.011	0.840	0.618	0.430
jsoup-78aeac18c6	CharacterReader	0.012	0.835	0.009	0.850	0.006	0.865
jsoup-9170b1d17b	Attributes	0.006	0.130	0.424	0.390	0.075	0.260
jsoup-b129bc9e3b	HttpConnection	0.037	0.780	0.004	0.885	0.287	0.645
jsoup-c507588b5c	TextNode	0.559	0.420	0.906	0.480	0.036	0.225
jsoup-c61ce94f35	StructuralEvaluator	0.584	0.450	0.803	0.530	1.000	0.495
jsoup-e1880ad73e	UrlBuilder	1.000	0.500	1.000	0.500	0.583	0.450
jsoup-eff15210b0	HttpConnection	0.003	0.890	0.004	0.880	0.001	0.925
jsoup-f0eb6bd1cc	UrlBuilder	0.301	0.600	0.301	0.600	0.651	0.550
markedj-2881d5b547	Marked	0.871	0.475	0.434	0.600	0.062	0.725
nfe-ec5ddf7e73	MDFInfoModalRodoviarioVeiculoReboque	1.000	0.500	1.000	0.500	1.000	0.500
semver4j-48ffbfd1f6	XRangeProcessor	0.007	0.815	0.144	0.650	0.004	0.850
semver4j-bf853ab269	Semver	0.129	0.700	0.073	0.735	0.005	0.860
traccar-074dc016d2	MediaFilter	1.000	0.500	1.000	0.500	1.000	0.500
traccar-1a1126d2d3	HuaShengProtocolDecoder	1.000	0.500	1.000	0.500	1.000	0.500
traccar-28440b7726	RuptelaProtocolDecoder	1.000	0.500	1.000	0.500	1.000	0.500
traccar-2ac77554f7	T55ProtocolDecoder	0.815	0.470	0.375	0.400	0.963	0.510
traccar-2dd48fa51b	GatorProtocolEncoder	1.000	0.500	0.368	0.450	1.000	0.500
traccar-3ba077b000	TramigoFrameDecoder	0.248	0.655	0.484	0.595	0.393	0.615
traccar-532c414196	WatchFrameDecoder	0.640	0.440	0.671	0.445	0.522	0.580
traccar-596036dc33	Jt600FrameDecoder	1.000	0.495	0.789	0.540	0.193	0.325
traccar-6e5481ebb1	SuntechProtocolDecoder	1.000	0.500	1.000	0.500	1.000	0.500
traccar-7ade92a97f	LaipacProtocolDecoder	0.681	0.450	0.451	0.585	1.000	0.500
traccar-94fbc93f8b	GalileoFrameDecoder	0.093	0.715	0.196	0.670	0.432	0.605
traccar-9a427527da	Minifinder2ProtocolEncoder	0.368	0.550	1.000	0.500	1.000	0.500
traccar-a07e078645	Tk103ProtocolDecoder	0.452	0.600	0.514	0.585	0.269	0.645
traccar-a24d7d5d7a	FreematicsProtocolDecoder	0.278	0.640	0.526	0.585	0.254	0.645
traccar-ab801e8565	Tk103ProtocolDecoder	0.055	0.750	0.285	0.630	0.668	0.555
traccar-b3c6e22fc1	BufferUtil	0.038	0.245	0.504	0.420	0.303	0.375
traccar-b5d5ec4318	Gt06ProtocolDecoder	0.302	0.360	0.087	0.270	0.236	0.340
traccar-ec2b7b64a8	Parser	0.005	0.855	0.036	0.765	0.157	0.680
traccar-f1470e5670	WatchFrameDecoder	0.502	0.410	0.507	0.410	0.641	0.435
traccar-f73263da48	WialonProtocolDecoder	1.000	0.500	1.000	0.500	1.000	0.500
word-wrap-930eb5e91a	WordWrap	0.472	0.600	0.570	0.580	0.704	0.555

Table 6.3: Table showing the Mann-Whitney U Tests p-value and Vargha-Delaney $\hat{A}_{12}$ effect size when comparing baseline to different seeding configurations w.r.t. mutation score. Highlighted p-value indicates a statistically significant difference from the baseline (p-value < 0.05), where a $\hat{A}_{12}$ effect size score greater than 0.5 shows better performance for the corresponding seeding configuration, while a lower score indicates better baseline performance.

6.1.4 Area Under The Curve

Lastly, we examine the result for the area under the curve shown in Table 6.4. It is possible that both the baseline and seeding configurations reach the same total number of targets; however, the time of when they arrived at that point may differ significantly thanks to the previously extracted information from the seed. This is an important metric because if the search algorithm can achieve the same or higher set of coverage goals during the search faster, then it could either terminate early in case of covering all objectives or spend more time attempting to cover even more objectives.

Once again, the table shows that in most of the benchmarks, 64%, there is no statistically significant difference between the baseline and any of the three seeding configurations. Despite that, AUC is still the metric where we see the most difference, with 17 benchmark classes, 14 of which show positive improvements for at least one seeding configuration. The three classes where we see the baseline perform better are `jsoup-c507588b5c` in cloning configuration, `traccar-b5d5ec4318` in carving configuration, and `traccar-f1470e5670` in combined configuration. Those results once again show that any one of the three seeding configurations could lose to a baseline configuration.

As for the positive performance of seeding configurations, we can see that eight classes have an improved score in all three configurations, indicating that in certain situations any seeding strategy will be sufficient to make some difference. The improvement in other six classes is distributed as follows among the configurations, combined configuration sees improvement in two of those classes (`java-solutions-7a73ea56d0` , `jsoup-4864af45af`), carving configuration sees improvement in two classes (`jsoup-4864af45af`, `traccar-2dd48fa51b`), and cloning configuration sees improvement in

(`crawler-commons-d8a6126365`, `formatter-maven-plugin-a6994326aa`, `java-solutions-7a73ea56d0`, `jsoup-23573ef31c`, `traccar-2dd48fa51b`) five of those classes. Although the standalone cloning configuration has the most classes which it positively impacts, it does not subsume all classes which improve in other standalone configurations, therefore, it cannot be considered best given the data.

Benchmark	File	Baseline vs Combined		Baseline vs Carving		Baseline vs Cloning	
		p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
ConfigMe-7449db4901	CommentsConfiguration	0.052	0.740	0.078	0.350	0.147	0.675
ConfigMe-cab40d1c3c	PropertyListBuilder	0.212	0.670	0.186	0.320	0.623	0.570
aws-java-nio-spi-for-s3-890fdddf30	S3Path	0.623	0.570	0.970	0.510	0.140	0.700
aws-secretsmanager-jdbc-42dc301cc7	AWSSecretsManagerPostgreSQLDriver	1.000	0.500	1.000	0.500	1.000	0.500
crawler-commons-d8a6126365	SimpleRobotRulesParser	0.273	0.650	0.076	0.740	0.045	0.770
crowdin-api-client-java-334d414753	JacksonJsonTransformer	0.003	0.900	0.004	0.890	0.002	0.910
epubcheck-8575a6b1c4	EpubChecker	0.054	0.760	0.385	0.620	0.427	0.610
formatter-maven-plugin-a6994326aa	CssFormatter	0.821	0.465	0.650	0.435	0.011	0.840
java-solutions-7a73ea56d0	OnlineStockSpan	0.007	0.840	0.456	0.580	0.001	0.920
java-stellar-sdk-06641953c4	KeyPair	0.791	0.540	0.678	0.560	0.345	0.370
java-stellar-sdk-d6379e9615	Transaction	0.521	0.590	0.850	0.530	0.850	0.530
jsoup-13f7ef9241	DataNode	0.014	0.830	0.005	0.880	0.026	0.800
jsoup-23573ef31c	Document	0.241	0.340	0.345	0.370	0.007	0.860
jsoup-23ea77ef4b	W3CDom	0.850	0.470	0.571	0.420	0.241	0.660
jsoup-4864af45af	Safelist	0.003	0.900	0.004	0.890	0.212	0.670
jsoup-78aeac18c6	CharacterReader	0.002	0.920	0.001	0.940	0.004	0.890
jsoup-9170b1d17b	Attributes	0.850	0.470	0.427	0.390	0.140	0.300
jsoup-b129bc9e3b	HttpConnection	0.054	0.760	0.162	0.690	0.140	0.700
jsoup-c507588b5c	TextNode	0.427	0.610	0.273	0.650	0.045	0.230
jsoup-c61ce94f35	StructuralEvaluator	0.734	0.450	0.850	0.470	0.140	0.300
jsoup-e1880ad73e	UrlBuilder	0.850	0.470	0.850	0.470	0.241	0.340
jsoup-eff15210b0	HttpConnection	0.0	0.980	0.002	0.920	0.001	0.930
jsoup-f0eb6bd1cc	UrlBuilder	0.054	0.760	0.241	0.660	0.970	0.510
markedj-2881d5b547	Marked	0.005	0.870	0.016	0.820	0.0	0.960
nfe-ec5ddf7e73	MDFInfoModalRodoviarioVeiculoReboque	0.003	0.900	0.0	1.000	0.0	1.000
semver4j-48ffbfd1f6	XRangeProcessor	0.385	0.620	0.521	0.410	0.623	0.570
semver4j-bf853ab269	Semver	0.002	0.910	0.003	0.900	0.017	0.820
traccar-074dc016d2	MediaFilter	0.345	0.630	0.385	0.620	0.734	0.450
traccar-1a1126d2d3	HuaShengProtocolDecoder	0.678	0.560	0.273	0.650	0.970	0.510
traccar-28440b7726	RuptelaProtocolDecoder	1.000	0.500	0.910	0.480	0.791	0.540
traccar-2ac77554f7	T55ProtocolDecoder	0.427	0.390	0.473	0.400	0.385	0.380
traccar-2dd48fa51b	GatorProtocolEncoder	0.104	0.720	0.038	0.780	0.017	0.820
traccar-3ba077b000	TramigoFrameDecoder	0.734	0.450	0.427	0.610	0.970	0.510
traccar-532c414196	WatchFrameDecoder	0.076	0.260	0.212	0.330	1.000	0.500
traccar-596036dc33	Jt600FrameDecoder	0.427	0.390	0.970	0.490	0.970	0.490
traccar-6e5481ebb1	SuntechProtocolDecoder	0.734	0.550	0.910	0.520	0.140	0.300
traccar-7ade92a97f	LaipacProtocolDecoder	0.734	0.450	0.850	0.470	0.678	0.440
traccar-94fbc93f8b	GalileoFrameDecoder	0.473	0.400	0.970	0.510	0.121	0.710
traccar-9a427527da	Minifinder2ProtocolEncoder	0.307	0.640	0.345	0.630	0.850	0.470
traccar-a07e078645	Tk103ProtocolDecoder	0.970	0.510	0.734	0.450	0.473	0.600
traccar-a24d7d5d7a	FreematicsProtocolDecoder	0.910	0.480	0.910	0.520	0.791	0.540
traccar-ab801e8565	Tk103ProtocolDecoder	0.521	0.590	0.910	0.520	0.970	0.490
traccar-b3c6e22fc1	BufferUtil	0.054	0.240	0.089	0.270	0.970	0.490
traccar-b5d5ec4318	Gt06ProtocolDecoder	0.140	0.300	0.031	0.210	0.734	0.550
traccar-ec2b7b64a8	Parser	0.001	0.940	0.001	0.940	0.003	0.900
traccar-f1470e5670	WatchFrameDecoder	0.038	0.220	0.054	0.240	0.910	0.480
traccar-f73263da48	WialonProtocolDecoder	0.910	0.520	0.473	0.400	0.734	0.450
word-wrap-930eb5e91a	WordWrap	1.000	0.500	0.623	0.430	0.473	0.600

Table 6.4: Table showing the Mann-Whitney U Tests p-value and Vargha-Delaney $\hat{A}_{12}$ effect size when comparing baseline to different seeding configurations w.r.t. normalized area under the curve (AUC). Highlighted p-value indicates a statistically significant difference from the baseline (p-value < 0.05), where a $\hat{A}_{12}$ effect size score greater than 0.5 shows better performance for the corresponding seeding configuration, while a lower score indicates better baseline performance.

6.2 Discussion

LLM Limitations

As described in Chapter 4, our approach, on a high level, can be split into two parts. The first is the generation of LLM-based test cases through TestSpark. The second is EvoSuite-based test case generation, where we supply compiled LLM-based test cases as a seed. In order for our two-part approach to work properly, it is important that the first LLM-based part manages to produce tests that can be compiled. If it does not succeed, then our approach cannot benefit from the seeding and becomes equivalent to the default test case generation approach. Hence, the first thing we did is check how well the ChatGPT-4o model was generating test cases for our benchmarks. We can break up this analysis into two parts; the first part looks at the number of cases where the LLM could not generate any cases at all, Figure 6.1. The second part will look at the distribution of cases where LLM-generated tests could not be compiled, Figure 6.2.

From Figure 6.1 we can see that there are 16 benchmark projects for which ChatGPT-4o could not generate any tests for the ten iterations. The most likely reason for such a failure is the context-size limitations of the LLM. It could be that either the query or the response repeatedly exceeds the size limits, leading to an empty or unfinished response. A slightly more interesting observation from the same figure is that there are seven projects for which the LLM struggled to generate test cases for only one out of ten iterations. The query size is an unlikely culprit for this failure, because it should remain the same across all iterations and, as is evident from the data, it was not a problem for the other nine iterations. Therefore, the most likely reason is the response size limitation of ChatGPT-4o. If the response of a model exhausts the available response token amount, then it might not be able to finish the code snippet, leading to difficulties in parsing and consequently discarding of the entire response. Exclusion of all of those benchmarks shrinks our dataset from 136 projects to 113.

Figure 6.2 further shows how ChatGPT-4o struggles with consistently producing a test that can be compiled and used. The histogram places all projects that have zero compiled tests despite having LLM generate some tests for them into different bins, representing the number of iterations where tests could not be compiled. From the figure, we can see that there is more spread in the histogram closer to the lower values of failure, i.e. the model mostly fails to produce functional tests for one or two iterations out of ten. Nevertheless, adding up all the failures gives us another 64 benchmarks which can be excluded from the dataset. It should be noted that one benchmark. `traccar-b083371beb`, is already accounted for in the histogram one because it also has one iteration where LLM could not generate any tests. Thus, our data set size decreases by 63 projects, leading to only 50 projects remaining in which LLM managed to consistently supply EvoSuite with functional seeds.

In total, we have excluded about 63% of the benchmark projects due to the ChatGPT-4o instability in the generation of functional tests. The important takeaway we can draw from the analysis of the first figure, 6.1, the one that examines the number of classes where LLM did not generate any tests, is that the prompt and response sizes are important. TestSpark,

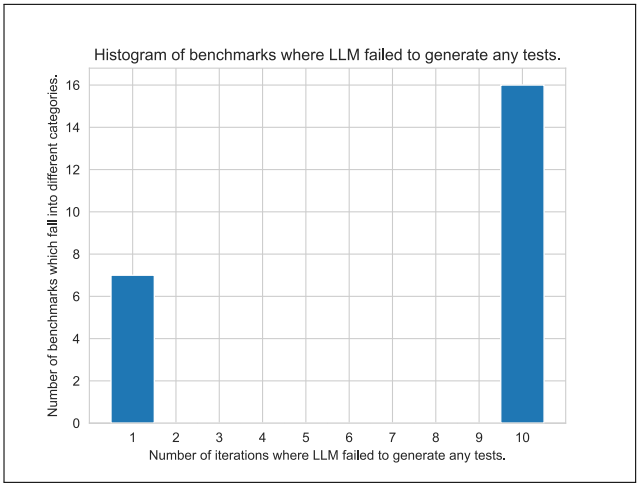


Figure 6.1: A histogram showing for how many benchmark iterations ChatGPT-4o struggled to generate tests. The x-axis is the number of iterations for which a benchmark failed to generate the LLM test. The y-axis is the number of benchmarks that fall into the bins defined on x-axis.

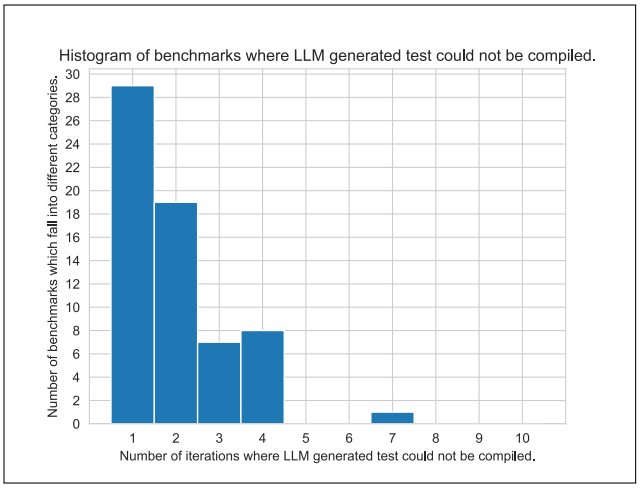


Figure 6.2: A histogram showing for how many benchmark iterations ChatGPT-4o generated testc could not be compiled. The x-axis is the number of iterations for which a benchmark could not compile LLM tests. The y-axis is the number of benchmarks that fall into the bins defined on x-axis.

the programme we use to generate LLM-based test cases, already attempts to dynamically adjust LLM's context size. However, even with this adjustment, we can see several classes where the prompt size is too large. Thus, suggesting more research in a more precise context collection approach for LLM prompts building for test generation purposes. Another takeaway is that LLM's response size could also be a problem and that perhaps the LLM prompt should not ask for a generation of a whole test suite but rather individual tests. Since it would be more resource efficient to ask for one test, it would be better to ask for one. case and get it, rather than asking for more at once and getting nothing.

The second figure, 6.2, the one that examines the case where LLM test cases could not be compiled, also points us in the direction of better prompt creation or an improved feedback loop, although in a less clear way. TestSpark already incorporates a feedback loop which attempts to fix compilation errors by prompting the LLM to fix them. Although it is not clear from our data how much this loop improved the test case generation performance, it is possible that it has greatly improved the number of functional test cases. Perhaps a further investigation into the feedback mechanism, the most often compilation failure reasons and prompt building could further make LLM-based test case generation more stable.

6.2.1 Overall Observations

When examining the four different tables (6.1, 6.2, 6.3, 6.4), corresponding to four different metrics, the results appear to show that for most classes, the seeding configurations do not improve the test generated by EvoSuite. After all, the highest number of individual benchmark classes that are influenced by any seeding configuration for a specific metric was 17 out of 48 (35%) for the AUC metric. However, if we count the number of unique benchmark classes which are influenced by any seeding configuration w.r.t. to any metric, then we will see that exactly half of the projects, 24 projects out of 48, are actually impacted by the seeding configurations. Furthermore, 18 of those 24 projects actually show a positive effect on some metrics, while only 8 of the 24 show negative effects of seeding configurations. The reason why the number of negative and positive projects does not add up to 24 ($18 + 8 = 26 \neq 24$) is because we have two projects (`jsoup-13f7ef9241`, `jsoup-23573ef31c`) where for some metrics we have improvement, while for others we have regression. The fact that `jsoup-13f7ef9241` experiences both an improvement in AUC in the cloning configuration and a regression in the mutation score in the same configuration is not contradictory. This is because the AUC is calculated based on the goals covered during the search process. Those goals are not directly related to the mutation score, as it is computed later on in the already generated test suite. The algorithm could have reached the same number of goals faster, but still results in worse assertions, since mutation is not one of the default criteria, hence leading to a worse mutation score. Furthermore, the divergent impact on the `jsoup-23573ef31c` benchmark is also not unexplainable. This is because improvement and regression occur in different configurations, namely the AUC is improved in the cloning configuration, while both branch coverage and mutation scores regress in the combined configuration.

However, the fact that our three different configurations influence half of the benchmark classes does not validate our LLM-based seeding approach, because we cannot recommend

to users running all three configurations instead of a single baseline. So, how did the different configurations perform individually? To answer this question, we can count the number of unique benchmark classes with statistically significant results for each particular configuration.

The combined configuration in total statistically significantly impacts 19 benchmark classes, 14 of which are positive and five are negative. The Carving Configuration in total impacts 14 classes, where 12 of those are positive while only two are negative. Lastly, the cloning configuration influences 15 benchmark classes, where 14 of them have a positive impact, while only two show a negative effect. Note that one of the classes in the cloning configuration with a negative impact is `jsoup-13f7ef9241` in the mutation score, but a positive one in the AUC. As explained in a couple of the paragraphs above, this is not an impossible situation to be in, given the independence of AUC and mutation score metrics.

Overall, grouping the impact on benchmark classes per configuration allows us to see that in most cases if seeding makes a difference, it is positive. Both the cloning and the combined configurations share the most classes with positive impact (14), but cloning seems to negatively affect only two and not five classes like the combined configuration.

From these data, one could suggest that using any of the three seeding configurations should provide better or the same results in most cases if LLM-generated tests are available. However, given poor generalization of this approach to other benchmark classes highlighted by the fact that only half of the remaining benchmarks (24/48) see any influence and the fact that we already excluded 88 benchmark classes out of 136 which at any point failed on the LLM test case generation step, it does not seem like a very certain endeavour. In general, if the developer has the resources to attempt test generation with LLM's for their one specific class, and it produces a set of functional tests, then the developer can also provide those tests as a seed and use one of the three configurations without expecting much of a negative influence but perhaps some positive one.

Nevertheless, our study is not over, as an intriguing question arises from all of that data. Why do we see a statistically significant difference from the baseline in half of the classes but not in the other? The statistically significant data we collected by comparing the baseline to different configuration w.r.t. different metrics inform us of whether the results of a configuration are statistically different from the baseline or whether we can get them by chance. We see that in half of the cases, the information contained within the provided seed influenced the search, mostly positively, but with occasional negative side effects. However, in the other half of the cases, there was no difference with the baseline? Why? What was the difference between the seed and the information it provided and how was this information not improving or hindering the search?

6.2.2 EvoSuite's Carving And Cloning Limitations

This section looks into the questions established a couple of the lines above, namely why there is no statistical difference for seeding configurations for half of the benchmarks.

To answer this question, we can start by looking at the information contained within the EvoSuite log files for each benchmark class generation. Those files contain quite a lot of information, although they can get pretty large and chaotic due to the reported warnings or

errors. However, all run configurations that use any type of seeding, cloning, carving, or combination should have the following entries in their logs, as shown in Figure 6.3.

```
...
* Executing tests from 28 test classes for carving
...
 -> Carved 24 tests for class class org.stellar.sdk.KeyPair from existing JUnit tests
 -> Carved 12 tests for class class net.i2p.crypto.eddsa.Utils ...
 -> Carved 2 tests for class class net.i2p.crypto.eddsa.math.Field ...
...
* Using 24 carved tests from existing JUnit tests for seeding
...
```

Figure 6.3: Example of seeding related entries in EvoSuite.log for the KeyPair class of java-stellar-sdk-06641953c4 benchmark.

The EvoSuite log files as shown in Figure 6.3 usually start with some information about the current run, which we truncated because it is not very relevant for seeding. The first interesting information for us is the total number of tests executed from the test classes for carving. This number, in the figure's example it is 28, is the total number of tests which are supplied to the EvoSuite for processing as a seed. After that, EvoSuite proceeds to attempt to extract information from the tests and create their representation within its system. If successful, EvoSuite will report the results for which classes it managed to carve objects and how many carved tests it can use for cloning. For example, in the figure, we have `-> Carved 24 tests for class class org.stellar.sdk.KeyPair from existing JUnit tests` line which implies that 24 of the 28 tests supplied were useful for the class `KeyPair` and were saved in the carved object pool if the carving option is enabled. If the carving option is not enabled, then the carving object pool would not be used, but this information could still be used for cloning. Whether the seed contains information useful for cloning is seen in the last line shown `* Using 24 carved tests from existing JUnit tests for seeding` . This states that 24 carved tests were found by the carving mechanism to contain useful information for the cloning, and so they could be used in the cloning process,

Going back to our question, what this information in the log files tells us is how many of the supplied tests are actually useful for the carving mechanism and the cloning mechanism. If the supplied tests were not useful at all, then we would not see lines such as `-> Carved 24 tests for class class org.stellar.sdk.KeyPair from existing JUnit tests` and `* Using 24 carved tests from existing JUnit tests for seeding` the search performance would be equivalent to the baseline configuration.

To extract all this information from the log files, we created a Python script which matches on those string patterns and extracts the relevant data. Additionally, the script also records Boolean values if cloning or carving did happen. For example, if a line similar to `* Using 24 carved tests from existing JUnit tests for seeding` is present, then we extract the number 24 from it and record that cloning did happen, otherwise cloning did not happen. Similar goes for the carving, but with the lines like `-> Carved 24 tests for class class org.stellar.sdk.KeyPair from existing JUnit tests`. If lines

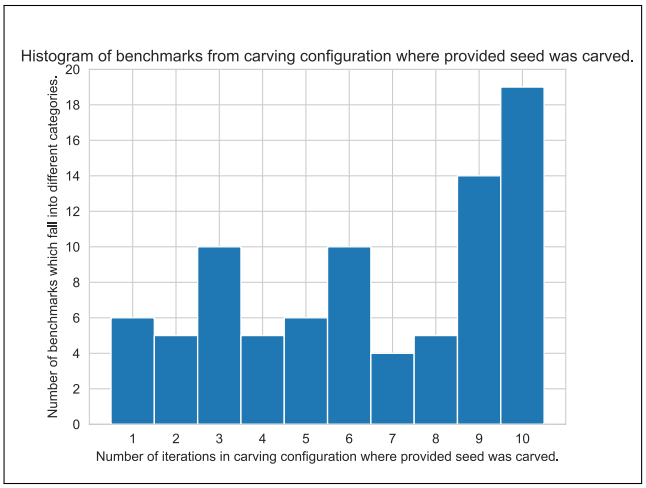


Figure 6.4: Histogram showing the number of benchmarks from carving configuration which fall into different bins (y-axis) where bins are the amount of iteration where carving mechanism was actully used (x-axis).

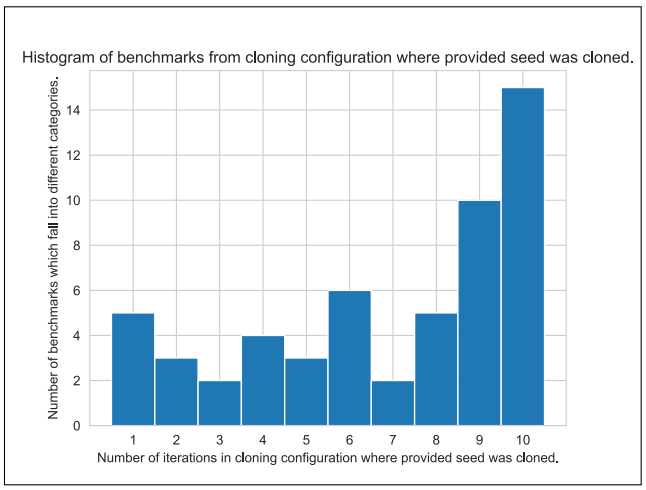


Figure 6.5: Histogram showing the number of benchmarks from cloning configuration which fall into different bins (y-axis) where bins are the amount of iteration where cloning mechanism was actully used (x-axis).

similar to that are present, then we consider carving to have happened and extract the information regarding how many tests were carved for which classes. To our great surprise, we get the results in Figures 6.4, 6.5, 6.6.

The results in Figure 6.4 show the number of benchmarks of the carving configuration which fall into different bins (y-axis) where the bins are the amount of iteration in which the carving mechanism was used actively (x-axis). From it we can see that the carving mechanism actually worked in all ten iterations only for 19 cases and not 48 benchmark cases, like we evaluated. This means that only in those 19 cases the performance of the

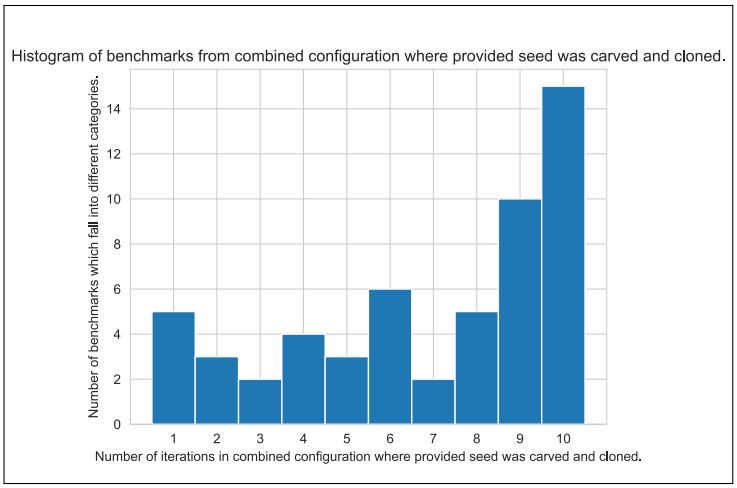


Figure 6.6: Histogram showing the number of benchmarks from combined configuration which fall into different bins (y-axis) where bins are the amount of iteration where carving and cloning mechanism was actully used (x-axis).

search was using the seed information; in the other cases it essentially defaulted to the baseline configuration.

Similar results can be seen in other figures. For example, Figure 6.5 shows the number of benchmarks of the cloning configuration that fall into different bins (y-axis) where bins are the amount of iteration where the cloning mechanism was actively used (x-axis). In it we can see that actually only in 15 cases the cloning mechanism used the seed information across all ten iterations.

While Figure 6.6 shows the number of benchmarks from the combined configuration that fall into different bins (y-axis) where bins are the amount of iteration in which the carving and cloning mechanism was used actively (x-axis). It also shows that only in 15 cases the cloning and carving mechanisms used the seed information across all ten iterations.

This finding sheds some light on why half of the classes have significant differences compared to the baseline and others do not. It was because there was actually no difference since the seeding mechanism, whether carving, cloning, or both, has struggled to get anything out of those seeds provided.

Note that the above tables have only 14 benchmark classes, not the 15 which overlap because `traccar-fb83a98d33` did not manage to have 10 successful iterations in the baseline configuration.

If we count the number of statistically significant results per configuration, then we get the following data. The carving configuration in total had eight classes where we saw statistical significance, and all eight of them were showing that the carving configuration performed better than the baseline. The cloning configuration follows the carving configuration with nine classes showing a statistically significant result, one of which showed both a positive and negative impact compared to the baseline. Lastly, the combined configuration influenced the most benchmark classes, 11, only one of which showed better performance

Benchmark	File	Baseline vs Combined		Baseline vs Carving		Baseline vs Cloning	
		p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
ConfigMe-cab40d1c3c	PropertyListBuilder	0.408	0.400	0.204	0.350	0.083	0.300
formatter-maven-plugin-a6994326aa	CssFormatter	1.000	0.500	1.000	0.500	1.000	0.500
java-solutions-7a73ea56d0	OnlineStockSpan	1.000	0.500	1.000	0.500	1.000	0.500
java-stellar-sdk-06641953c4	KeyPair	0.328	0.615	1.000	0.505	0.871	0.520
jsoup-13f7ef9241	DataNode	1.000	0.500	1.000	0.500	1.000	0.500
jsoup-4864af45af	Safelist	0.061	0.740	0.015	0.815	0.606	0.430
jsoup-78aeac18c6	CharacterReader	0.001	0.900	0.001	0.900	0.004	0.850
jsoup-9170b1d17b	Attributes	0.006	0.140	0.429	0.400	0.108	0.295
jsoup-eff15210b0	HttpConnection	0.001	0.945	0.0	0.965	0.002	0.905
markedj-2881d5b547	Marked	1.000	0.500	1.000	0.500	1.000	0.500
nfe-ec5ddf7e73	MDFInfoModalRodoviarioVeiculoReboque	1.000	0.500	1.000	0.500	1.000	0.500
semver4j-48ffbfd1f6	XRangeProcessor	0.033	0.760	0.023	0.780	0.001	0.915
traccar-ec2b7b64a8	Parser	0.005	0.855	0.012	0.810	0.021	0.775
word-wrap-930eb5e91a	WordWrap	0.035	0.765	0.938	0.515	0.443	0.395

Table 6.5: Line Coverage

Benchmark	File	Baseline vs Combined		Baseline vs Carving		Baseline vs Cloning	
		p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
ConfigMe-cab40d1c3c	PropertyListBuilder	0.408	0.400	0.204	0.350	0.083	0.300
formatter-maven-plugin-a6994326aa	CssFormatter	1.000	0.500	1.000	0.500	1.000	0.500
java-solutions-7a73ea56d0	OnlineStockSpan	1.000	0.500	1.000	0.500	1.000	0.500
java-stellar-sdk-06641953c4	KeyPair	0.226	0.640	0.670	0.460	0.871	0.520
jsoup-13f7ef9241	DataNode	1.000	0.500	1.000	0.500	1.000	0.500
jsoup-4864af45af	Safelist	0.062	0.750	0.011	0.840	0.789	0.460
jsoup-78aeac18c6	CharacterReader	0.005	0.860	0.002	0.910	0.004	0.880
jsoup-9170b1d17b	Attributes	0.005	0.130	0.814	0.465	0.174	0.320
jsoup-eff15210b0	HttpConnection	0.190	0.670	0.429	0.605	0.345	0.625
markedj-2881d5b547	Marked	1.000	0.500	1.000	0.500	1.000	0.500
nfe-ec5ddf7e73	MDFInfoModalRodoviarioVeiculoReboque	1.000	0.500	1.000	0.500	1.000	0.500
semver4j-48ffbfd1f6	XRangeProcessor	0.033	0.760	0.023	0.780	0.001	0.930
traccar-ec2b7b64a8	Parser	0.005	0.855	0.012	0.810	0.021	0.775
word-wrap-930eb5e91a	WordWrap	0.179	0.680	1.000	0.495	0.969	0.510

Table 6.6: Branch Coverage

in the baseline case.

There does not seem to be any single configuration that subsumes others. To further see the differences between the configurations, we can examine whether they cover the same parts.

To examine the coverage overlap between different configurations, we can look at the generated JaCoCo reports, which contain information of covered and missed branches and instructions per line. Precisely, we make a script that collects the unique line numbers of covered instructions and branches into a set, and we do this for each class and each configuration. Afterwards, we find set intersections between them to see how much of an overlap there is between different configurations.

First, we examine the 12 benchmark classes for which we see statistical signfincae in

Benchmark	File	Baseline vs Combined		Baseline vs Carving		Baseline vs Cloning	
		p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
ConfigMe-cab40d1c3c	PropertyListBuilder	0.408	0.400	0.204	0.350	0.064	0.280
formatter-maven-plugin-a6994326aa	CssFormatter	0.204	0.650	0.398	0.600	1.000	0.500
java-solutions-7a73ea56d0	OnlineStockSpan	1.000	0.500	1.000	0.500	1.000	0.500
java-stellar-sdk-06641953c4	KeyPair	0.307	0.620	1.000	0.500	1.000	0.505
jsoup-13f7ef9241	DataNode	0.137	0.320	0.398	0.400	0.032	0.250
jsoup-4864af45af	Safelist	0.150	0.695	0.011	0.840	0.618	0.430
jsoup-78aeac18c6	CharacterReader	0.012	0.835	0.009	0.850	0.006	0.865
jsoup-9170b1d17b	Attributes	0.006	0.130	0.424	0.390	0.075	0.260
jsoup-eff15210b0	HttpConnection	0.003	0.890	0.004	0.880	0.001	0.925
markedj-2881d5b547	Marked	0.871	0.475	0.434	0.600	0.062	0.725
nfe-ec5ddf7e73	MDFInfoModalRodoviarioVeiculoReboque	1.000	0.500	1.000	0.500	1.000	0.500
semver4j-48ffbfd1f6	XRangeProcessor	0.007	0.815	0.144	0.650	0.004	0.850
traccar-ec2b7b64a8	Parser	0.005	0.855	0.036	0.765	0.157	0.680
word-wrap-930eb5e91a	WordWrap	0.472	0.600	0.570	0.580	0.704	0.555

Table 6.7: Mutation Score

Benchmark	File	Baseline vs Combined		Baseline vs Carving		Baseline vs Cloning	
		p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
ConfigMe-cab40d1c3c	PropertyListBuilder	0.212	0.670	0.186	0.320	0.623	0.570
formatter-maven-plugin-a6994326aa	CssFormatter	0.821	0.465	0.650	0.435	0.011	0.840
java-solutions-7a73ea56d0	OnlineStockSpan	0.007	0.840	0.456	0.580	0.001	0.920
java-stellar-sdk-06641953c4	KeyPair	0.791	0.540	0.678	0.560	0.345	0.370
jsoup-13f7ef9241	DataNode	0.014	0.830	0.005	0.880	0.026	0.800
jsoup-4864af45af	Safelist	0.003	0.900	0.004	0.890	0.212	0.670
jsoup-78aeac18c6	CharacterReader	0.002	0.920	0.001	0.940	0.004	0.890
jsoup-9170b1d17b	Attributes	0.850	0.470	0.427	0.390	0.140	0.300
jsoup-eff15210b0	HttpConnection	0.0	0.980	0.002	0.920	0.001	0.930
markedj-2881d5b547	Marked	0.005	0.870	0.016	0.820	0.0	0.960
nfe-ec5ddf7e73	MDFInfoModalRodoviarioVeiculoReboque	0.003	0.900	0.0	1.000	0.0	1.000
semver4j-48ffbfd1f6	XRangeProcessor	0.385	0.620	0.521	0.410	0.623	0.570
traccar-ec2b7b64a8	Parser	0.001	0.940	0.001	0.940	0.003	0.900
word-wrap-930eb5e91a	WordWrap	1.000	0.500	0.623	0.430	0.473	0.600

Table 6.8: AUC Normalized

at least one of the metrics in one of the configurations. Among them, we have nine classes with a complete overlap in coverage. The only three classes which do not overlap are `jsoup-eff15210b0`, `semver4j-48ffbfd1f6` and `traccar-ec2b7b64a8`.

`jsoup-eff15210b0` combined configuration has three unique entries. Everything else is overlapped by all configurations.

`semver4j-48ffbfd1f6` cloning configuration has seven unique elements to itself. While combined and cloning configurations overlap in 32-line numbers but other configurations only in 26-line numbers.

`traccar-ec2b7b64a8` combined configuration has 10 unique elements, while carving and cloning share the biggest overlap of 72-line numbers, while other configs share 63 entries.

The two classes which did not have any significance are `ConfigMe-cab40d1c3c` and `java-stellar-sdk-06641953c4`.

`ConfigMe-cab40d1c3c` It appears that coverage overlap analysis shows that there is no difference between any of the seeding configurations and baseline. Therefore, there might be no statistical difference because the normal EvoSuite already has a pretty good performance without seeding for this class due to class-specific characteristics.

Results for `java-stellar-sdk-06641953c4`, show inconsistencies which in theory should have not happened. According to the data, different configurations extracted different amounts of information from the same set of test cases. This should not be the case since carving and cloning mechanisms are supposed to be deterministic, i.e. given the same set of tests, we should always extract the same information. For example, log files for `java-stellar-sdk-06641953c4` iteration four show that in the combined configuration there were 21 tests carved for `KeyPair` class and subsequently used for seeding. However, in the same iteration four but for the cloning configuration, the logs report that 24 tests were carved. Both configurations were provided the same set of tests on the same iteration. Upon attempting to replicate the results locally on a machine rather than a server, we did not see the same inconsistent behaviour.

Chapter 7

Conclusions & Future Work

7.1 Conclusion

The results and discussion presented in Chapter 6 regarding our research, "**What is the impact of ChatGPT-4o generated tests when supplied as a seed for carving and cloning strategies on EvoSuite's test case generation?**", lead us to the following conclusions and takeaways.

Our approach described in Chapter,4 which, on the high level, consists of first generating tests using the ChatGPT-4o model and then supplying them to EvoSuite as a seed, suffers a major loss in benchmarks due to LLM's instability of generating functional tests. The initial GitBug benchmark consisted of 136 Java 11 projects. However, our results show that ChatGPT-4o was able to reliably generate functional tests for all ten iterations of the evaluation only on 50 benchmarks. The other 86 benchmarks had at least one iteration in which an LLM could not complete the response due to context size limitations, or the given response contained code that could not be compiled. This 63% benchmark loss occurs despite the TestSpark attempts to reduce the prompt size and the feedback loop that prompt the LLM to fix the compilation issues. Those observations contribute to our research question by showing that the impact of the ChatGPT-4o test as a seed for EvoSuite can be non-existent because LLM produces an unusable by the tool response. Future research could examine how our approach could be changed with respect to prompt building, the chosen model, and the desired output to increase the consistency of information extracted from LLM's responses.

Apart from suffering a great loss in the number of benchmarks due to ChatGPT-4o's struggles to produce functional tests, our results show that there exists another loss of information related to the EvoSuite's seeding mechanism and the supplied tests. Specifically, according to the EvoSuite logs, the tool was unable to extract any information to use in the search process for other 29/33[1] benchmarks. This marks another discovered problem of our approach, which could be explored further in future work.

Having excluded all benchmark classes that failed for LLM or seed extraction reasons,

[1]The carving configuration worked in 19 cases while the cloning and carving worked only in 15 cases.

we get only 14[2] benchmark classes which successfully worked across all configurations. In general, we can see some improvement in line coverage, branch coverage, or mutation score in six out of 14 benchmark classes across all seeding configurations. Additionally, there are five classes which achieve the same line coverage for all iterations but improved areas under the curve scores, thereby indicating a faster convergence. In total, we can see positive improvements on 11 of 14 benchmarks. As for the negative improvement, there are only two such cases; one of them is `jsoup-9170b1d17b` which consistently performs better in the baseline configuration than in the combined configuration. The second case is `jsoup-13f7ef9241` which performed better w.r.t. mutation score in cloning configuration. There are only two classes which show complete indifference across all metrics, namely `ConfigMe-cab40d1c3c` and `java-stellar-sdk-06641953c4`. The former seems to have a complete overlap with respect to line coverage across all four configurations; the latter, according to the logs, appears to have some inconsistencies across the different configurations w.r.t. how much seeding has happened.

Furthermore, we observe little difference between the performance of different seeding configurations. The coverage overlap analysis shows a difference for only three classes with some wins for different configurations, i.e. no single configuration that always outperforms others.

To summarize our findings and answer the above-stated research question, we can see that LLM-based seeding can improve test case generation performance, with little drawbacks, w.r.t. to line coverage, branch coverage, mutation score, and area under the curve. No single seeding strategy, cloning, carving, or combination seem to consistently outperform other strategies, and many improvements seem to overlap across all three strategies. However, our results also show the limitations and areas for improvement for our LLM-based seeding approach. Almost 90% of the initial benchmark classes were lost due to the ChatGPT-4o's struggles at generating functional tests or EvoSuite's seeding mechanism not being able to extract any information out of the provided tests. Those limitations indicate areas for future research and caution for researchers and developers implementing similar approaches.

7.2 Future work

Our work has shown several limitations of our approach and led us to more unanswered questions that can be explored further. In this section, we will discuss some potential ideas which can be further researched in the future.

The first struggle our approach encountered was in LLM's ability to complete the reply with a set of test cases that can be compiled. That is the struggle that happened despite TestSpark's mechanism for reducing the prompt size and the feedback loop for asking LLM to fix compilation errors. Future research could investigate what should go into the prompt so that its size can be reduced to a manageable amount. Additionally, it would be beneficial to investigate what compilation failures are the most frequent and whether there are any

[2]One benchmark out of 15 which worked across all three seeding configurations is excluded because it failed in the baseline case.

prompt engineering techniques or other repair methods that could increase the amount of functional code extracted from LLM.

The second area of struggle was in EvoSuite's ability to extract information from the provided LLM-generated tests. From our results, it is not clear whether this happened due to the peculiarities of LLM-generated tests or specifics of the EvoSuite's seeding mechanism, or both. Therefore, it would be informative to examine the difference between the LLM tests generated from which EvoSuite successfully extracted the information and from which it did not manage it. This insight could be useful for improving the LLM prompt to make the response more specific to the needs of the seeding mechanism or to improve the seeding mechanism itself.

A new study with a larger benchmark and more consistent seeding information can be conducted to further examine the impact of seeding and similarities between seeding strategies, as our study managed to do that only in 14 classes. A bigger study could also introduce new configurations in which, in addition to the LLM-based test, it also investigates the performance of available human written tests as a seed. Those comparisons could give us data on the quality of the LLM-generated tests as a seed in contrast to the available human written ones.

Lastly, it would be interesting to see test case generation tools that are directly integrated with the LLMs, thereby allowing for more granular and precise querying. The current approach struggles with getting the information out of an LLM and into the EvoSuite's system, where this information is correctly represented. Instead, if LLM capabilities were to be directly integrated into EvoSuite or other test case generation tools, this would potentially allow for more precise LLM requests. For example, if EvoSuite identifies a need for a complex object, then it could decide to query the LLM rather than spend time searching for how to create this object through the iteration of an evolutionary algorithm. This could lead to better efficiency and effectiveness, as the LLM query would focus specifically on asking for this specific single object instead of a whole test suite.

Bibliography

[1] Structural testing and code coverage. In *Effective software testing: a developer's guide*. Manning Publications Co, Shelter Island, NY, first edition edition, 2022. ISBN 978-1-63835-058-3.

[2] A. M. Abdullin and V. M. Itsykson. Kex: A Platform For Analysis Of JVM ProgramsKex: JVM- | Request PDF. *ResearchGate*, October 2024. doi: 10.31799/1684-8853-2022-1-30-43. URL https://www.researchgate.net/publication/358990251_Kex_A_Platform_For_Analysis_Of_JVM_ProgramsKex_platforma_dla_analiza_JVM-programm.

[3] Azat Abdullin, Pouria Derakhshanfar, and Annibale Panichella. Test wars: A comparative study of sbst, symbolic execution, and llm-based approaches to unit test generation. *arXiv preprint arXiv:2501.10200*, 2025.

[4] Bushra Alhijawi and Arafat Awajan. Genetic algorithms: theory, genetic operators, solutions, and applications. *Evolutionary Intelligence*, 17(3):1245–1256, June 2024. ISSN 1864-5917. doi: 10.1007/s12065-023-00822-6. URL https://link.springer.com/article/10.1007/s12065-023-00822-6. Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 3 Publisher: Springer Berlin Heidelberg.

[5] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272, May 2017. doi: 10.1109/ICSE-SEIP.2017.27. URL https://ieeexplore.ieee.org/abstract/document/7965450.

[6] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):

1978–2001, August 2013. ISSN 01641212. doi: 10.1016/j.jss.2013.02.061. URL https://linkinghub.elsevier.com/retrieve/pii/S0164121213000563.

[7] Mauricio Aniche, Arie Deursen, and Steve Freeman. *Effective and systematic software testing*. Manning Publications Co, Shelter Island, NY, first edition edition, 2022. ISBN 978-1-63835-058-3.

[8] Andrea Arcuri. EvoMaster: Evolutionary Multi-context Automated System Test Generation. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 394–397, April 2018. doi: 10.1109/ICST.2018.00046. URL https://ieeexplore.ieee.org/abstract/document/8367066.

[9] Andrea Arcuri and Lionel Briand. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014. ISSN 1099-1689. doi: 10.1002/stvr.1486. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1486. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1486.

[10] Andrea Arcuri and Gordon Fraser. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, June 2013. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-013-9249-9. URL http://link.springer.com/10.1007/s10664-013-9249-9.

[11] Shuvayan Brahmachary, Subodh M. Joshi, Aniruddha Panda, Kaushik Koneripalli, Arun Kumar Sagotra, Harshil Patel, Ankush Sharma, Ameya D. Jagtap, and Kaushic Kalyanaraman. Large language model-based evolutionary optimizer: Reasoning with elitism. *Neurocomputing*, 622:129272, March 2025. ISSN 0925-2312. doi: 10.1016/j.neucom.2024.129272. URL https://www.sciencedirect.com/science/article/pii/S0925231224020435.

[12] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.

[13] Jinyu Cai, Jinglue Xu, Jialong Li, Takuto Yamauchi, Hitoshi Iba, and Kenji Tei. Exploring the Improvement of Evolutionary Computation via Large Language Models. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '24 Companion, pages 83–84, New York, NY, USA, August 2024. Association for Computing Machinery. ISBN 979-8-4007-0495-6. doi: 10.1145/3638530.3664086. URL https://dl.acm.org/doi/10.1145/3638530.3664086.

[14] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, pages 572–576, New York, NY, USA, July 2024. Association for Computing Machinery. ISBN 979-8-4007-0658-5. doi: 10.1145/3663529.3663801. URL https://dl.acm.org/doi/10.1145/3663529.3663801.

[15] Silei Cheng, Zhe Gan, Zhengyuan Yang, Shuohang Wang, Jianfeng Wang, Jordan Boyd-Graber, and Lijuan Wang. Prompting GPT-3 To Be Reliable. May 2023. URL https://www.microsoft.com/en-us/research/publication/prompting-gpt-3-to-be-reliable/.

[16] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. Effective test generation using pre-trained Large Language Models and mutation testing. *Information and Software Technology*, 171:107468, July 2024. ISSN 0950-5849. doi: 10.1016/j.infsof.2024.107468. URL https://www.sciencedirect.com/science/article/pii/S0950584924000739.

[17] Nicolas Erni, Mohammed Al-Ameen, Christian Birchler, Pouria Derakhshanfar, Stephan Lukasczyk, and Sebastiano Panichella. SBFT Tool Competition 2024 - Python Test Case Generation Track. In *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing*, SBFT '24, pages 37–40, New York, NY, USA, September 2024. Association for Computing Machinery. ISBN 979-8-4007-0562-5. doi: 10.1145/3643659.3643930. URL https://dl.acm.org/doi/10.1145/3643659.3643930.

[18] D.B. Fogel. What is evolutionary computation? *IEEE Spectrum*, 37(2):26–32, 2000. doi: 10.1109/6.819926.

[19] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0443-6. doi: 10.1145/2025113.2025179. URL http://doi.acm.org/10.1145/2025113.2025179.

[20] Gordon Fraser and Andrea Arcuri. Evolutionary Generation of Whole Test Suites. In *2011 11th International Conference on Quality Software*, pages 31–40, Madrid, Spain, July 2011. IEEE. ISBN 978-1-4577-0754-4. doi: 10.1109/QSIC.2011.19. URL http://ieeexplore.ieee.org/document/6004309/.

[21] Gordon Fraser and Andrea Arcuri. The Seed is Strong: Seeding Strategies in Search-Based Software Testing. In *Verification and Validation 2012 IEEE Fifth International Conference on Software Testing*, pages 121–130, April 2012. doi: 10.1109/ICST.2012.92. URL https://ieeexplore.ieee.org/document/6200103. ISSN: 2159-4848.

[22] Gordon Fraser and Andrea Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2), dec 2014. ISSN 1049-331X. doi: 10.1145/2685612. URL https://doi.org/10.1145/2685612.

[23] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with EvoSuite. *Empirical Software Engineering*, 20(3):611–639, June 2015. ISSN 1573-7616. doi: 10.1007/s10664-013-9288-2. URL

https://link.springer.com/article/10.1007/s10664-013-9288-2. Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 3 Publisher: Springer US.

[24] Alessio Gambi, Gunel Jahangirova, Vincenzo Riccio, and Fiorella Zampetti. Sbst tool competition 2022. In *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)*, pages 25–32, 2022. doi: 10.1145/3526072.3527538.

[25] Alessio Gambi, Gunel Jahangirova, Vincenzo Riccio, and Fiorella Zampetti. SBST tool competition 2022. In *Proceedings of the 15th Workshop on Search-Based Software Testing*, SBST '22, pages 25–32, New York, NY, USA, February 2023. Association for Computing Machinery. ISBN 978-1-4503-9318-8. doi: 10.1145/3526072.3527538. URL https://dl.acm.org/doi/10.1145/3526072.3527538.

[26] Erik Hemberg, Stephen Moskal, and Una-May O'Reilly. Evolving code with a large language model. *Genetic Programming and Evolvable Machines*, 25(2):21, September 2024. ISSN 1573-7632. doi: 10.1007/s10710-024-09494-2. URL https://doi.org/10.1007/s10710-024-09494-2.

[27] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 560–564, March 2015. doi: 10.1109/SANER.2015.7081877. URL https://ieeexplore.ieee.org/document/7081877. ISSN: 1534-5351.

[28] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931, 2023. doi: 10.1109/ICSE48619.2023.00085.

[29] Fei Liu, Xi Lin, Shunyu Yao, Zhenkun Wang, Xialiang Tong, Mingxuan Yuan, and Qingfu Zhang. Large Language Model for Multiobjective Evolutionary Optimization. In *Evolutionary Multi-Criterion Optimization*, pages 178–191. Springer, Singapore, 2025. ISBN 978-981-96-3538-2. doi: 10.1007/978-981-96-3538-2_13. URL https://link.springer.com/chapter/10.1007/978-981-96-3538-2_13. ISSN: 1611-3349.

[30] Shengcai Liu, Caishun Chen, Xinghua Qu, Ke Tang, and Yew-Soon Ong. Large Language Models as Evolutionary Optimizers. In *2024 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, June 2024. doi: 10.1109/CEC60901.2024.10611913. URL https://ieeexplore.ieee.org/document/10611913.

[31] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *ICSE'22: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172. ACM, 2022. doi: 10.1145/3510454.3516829.

[32] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. An empirical study of automated unit test generation for Python. *Empirical Softw. Engg.*, 28(2), January 2023. ISSN 1382-3256. doi: 10.1007/s10664-022-10248-w. URL https://doi.org/10.1007/s10664-022-10248-w.

[33] Marcos Macedo, Yuan Tian, Filipe Cogo, and Bram Adams. Exploring the Impact of the Output Format on the Evaluation of Large Language Models for Code Translation. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, FORGE '24, pages 57–68, New York, NY, USA, June 2024. Association for Computing Machinery. ISBN 979-8-4007-0609-7. doi: 10.1145/3650105.3652301. URL https://dl.acm.org/doi/10.1145/3650105.3652301.

[34] A.P. Mathur. *Foundations of Software Testing, 2/e*. Pearson Education India, 2013. ISBN 978-93-325-1765-3. URL https://books.google.de/books?id=OUA8BAAAQBAJ.

[35] Seokhyeon Moon and Yoonchan Jhi. EvoFuzz at the SBFT 2024 Tool Competition. In *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing*, SBFT '24, pages 63–64, New York, NY, USA, September 2024. Association for Computing Machinery. ISBN 979-8-4007-0562-5. doi: 10.1145/3643659.3648556. URL https://dl.acm.org/doi/10.1145/3643659.3648556.

[36] Clint Morris, Michael Jurado, and Jason Zutty. LLM Guided Evolution - The Automation of Models Advancing Models. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '24, pages 377–384, New York, NY, USA, July 2024. Association for Computing Machinery. ISBN 979-8-4007-0494-9. doi: 10.1145/3638529.3654178. URL https://dl.acm.org/doi/10.1145/3638529.3654178.

[37] Nadim Nachar. The Mann-Whitney U: A Test for Assessing Whether Two Independent Samples Come from the Same Distribution. *Tutorials in Quantitative Methods for Psychology*, 4(1):13–20, March 2008. ISSN 1913-4126. doi: 10.20982/tqmp.04.1.p013. URL http://www.tqmp.org/RegularArticles/vol04-1/p013.

[38] Mitchell Olsthoorn, D.M. Stallenberg, A. van Deursen, and A. Panichella. SynTest-Solidity: Automated Test Case Generation and Fuzzing for Smart Contracts. *The 44th International Conference on Software Engineering - Demonstration Track*, pages 202–206, 2022. doi: 10.1109/ICSE-Companion55297.2022.9793754. URL https://github.com/syntest-framework. Publisher: IEEE / ACM.

[39] Wendkuuni C. Ouedraogo, Kader Kabore, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawende F. Bissyande. LLMs and Prompting for Unit Test Generation: A Large-Scale Evaluation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pages 2464–2465, New York, NY, USA, October 2024. Association for Computing Machinery.

ISBN 979-8-4007-1248-7. doi: 10.1145/3691620.3695330. URL https://dl.acm.org/doi/10.1145/3691620.3695330.

[40] Wendkûuni C. Ouédraogo, Laura Plein, Kader Kaboré, Andrew Habib, Jacques Klein, David Lo, and Tegawendé F. Bissyandé. Enriching automatic test case generation by extracting relevant test inputs from bug reports. *Empirical Software Engineering*, 30 (3):1–60, May 2025. ISSN 1573-7616. doi: 10.1007/s10664-025-10635-z. URL https://link.springer.com/article/10.1007/s10664-025-10635-z. Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 3 Publisher: Springer US.

[41] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating Branch Coverage as a Many-Objective Optimization Problem. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, April 2015. doi: 10.1109/ICST.2015.7102604. URL https://ieeexplore.ieee.org/document/7102604. ISSN: 2159-4848.

[42] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2018. doi: 10.1109/TSE.2017.2663435.

[43] Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. SBST Tool Competition 2021. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 20–27, May 2021. doi: 10.1109/SBST52555.2021.00011. URL https://ieeexplore.ieee.org/abstract/document/9476243.

[44] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter Six - Mutation Testing Advances: An Analysis and Survey. In Atif M. Memon, editor, *Advances in Computers*, volume 112, pages 275–378. Elsevier, January 2019. doi: 10.1016/bs.adcom.2018.03.015. URL https://www.sciencedirect.com/science/article/pii/S0065245818300305.

[45] Juan Altmayer Pizzorno and Emery D. Berger. CoverUp: Coverage-Guided LLM-Based Test Generation, February 2025. URL http://arxiv.org/abs/2403.16218. arXiv:2403.16218 [cs].

[46] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability*, 26(5):366–401, 2016. ISSN 1099-1689. doi: 10.1002/stvr.1601. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1601. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1601.

[47] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM. *Proc. ACM Softw. Eng.*, 1

(FSE):43:951–43:971, July 2024. doi: 10.1145/3643769. URL https://dl.acm.org/doi/10.1145/3643769.

[48] Rustam Sadykov, Azat Abdullin, and Marat Akhin. Evokex at the SBFT 2024 Tool Competition. In *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing*, SBFT '24, pages 67–68, New York, NY, USA, September 2024. Association for Computing Machinery. ISBN 979-8-4007-0562-5. doi: 10.1145/3643659.3648558. URL https://dl.acm.org/doi/10.1145/3643659.3648558.

[49] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, January 2024. ISSN 1939-3520. doi: 10.1109/TSE.2023.3334955. URL https://ieeexplore.ieee.org/document/10329992.

[50] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 201–211, November 2015. doi: 10.1109/ASE.2015.86. URL https://ieeexplore.ieee.org/document/7372009.

[51] Philipp Straubinger, Marvin Kreis, Stephan Lukasczyk, and Gordon Fraser. Mutation Testing via Iterative Large Language Model-Driven Scientific Debugging, March 2025. URL http://arxiv.org/abs/2503.08182. arXiv:2503.08182 [cs].

[52] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation. *IEEE Transactions on Software Engineering*, 50(6):1340–1359, June 2024. ISSN 1939-3520. doi: 10.1109/TSE.2024.3382365. URL https://ieeexplore.ieee.org/abstract/document/10485640.

[53] AndrĂ¡s Vargha and Harold D. Delaney. A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000. ISSN 1076-9986. doi: 10.2307/1165329. URL https://www.jstor.org/stable/1165329. Publisher: [American Educational Research Association, Sage Publications, Inc., American Statistical Association].

[54] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pages 1258–1268, New York, NY, USA, October 2024. Association for Computing Machinery. ISBN 979-8-4007-1248-7. doi: 10.1145/3691620.3695501. URL https://dl.acm.org/doi/10.1145/3691620.3695501.

[55] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1556–1560, Virtual Event USA, November 2020. ACM. ISBN 978-1-4503-7043-1. doi: 10.1145/3368089.3417943. URL https://dl.acm.org/doi/10.1145/3368089.3417943.

[56] Xingyu Wu, Sheng-Hao Wu, Jibin Wu, Liang Feng, and Kay Chen Tan. Evolutionary Computation in the Era of Large Language Model: Survey and Roadmap. *IEEE Transactions on Evolutionary Computation*, 29(2):534–554, April 2025. ISSN 1941-0026. doi: 10.1109/TEVC.2024.3506731. URL https://ieeexplore.ieee.org/document/10767756.

[57] Danni Xiao, Yimeng Guo, Yanhui Li, and Lin Chen. Optimizing Search-Based Unit Test Generation with Large Language Models: An Empirical Study. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, Internetware '24, pages 71–80, New York, NY, USA, July 2024. Association for Computing Machinery. ISBN 979-8-4007-0705-6. doi: 10.1145/3671016.3674813. URL https://dl.acm.org/doi/10.1145/3671016.3674813.

[58] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. On the Evaluation of Large Language Models in Unit Test Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pages 1607–1619, New York, NY, USA, October 2024. Association for Computing Machinery. ISBN 979-8-4007-1248-7. doi: 10.1145/3691620.3695529. URL https://dl.acm.org/doi/10.1145/3691620.3695529.

[59] Ruofan Yang, Xianghua Xu, and Ran Wang. LLM-enhanced evolutionary test generation for untyped languages. *Automated Software Engineering*, 32(1):1–37, May 2025. ISSN 1573-7535. doi: 10.1007/s10515-025-00496-7. URL https://link.springer.com/article/10.1007/s10515-025-00496-7. Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 1 Publisher: Springer US.

[60] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997. ISSN 0360-0300. doi: 10.1145/267580.267590. URL https://dl.acm.org/doi/10.1145/267580.267590.