An LLM-Based Framework for Automated Python Test Generation and Mutation Testing Evaluation

by

Sadnan Nafis
21201249
Abdullah Al Walid
21201651
Amatus Subhan Anuja
24141126

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfilment of the requirements for the degree of
B.Sc. in Computer Science

Department of Computer Science and Engineering
Brac University
June 2025

Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:

Sadnan

Sadnan Nafis
21201249

Walid

Abdullah Al Walid
21201651

Amatus Subhan

Amatus Subhan Anuja
24141126

Approval

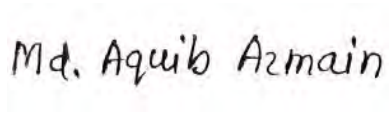
The thesis titled "An LLM-Based Framework for Automated Python Test Generation and Mutation Testing Evaluation" submitted by

1. Sadnan Nafis (21201249)
2. Abdullah Al Walid (21201651)
3. Amatus Subhan Anuja (24141126)

Of Spring, 2025 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science on June 20, 2025.

Examining Committee:

Supervisor:
(Member)



Md. Aquib Azmain
Lecturer
Department of Computer Science and Engineering
Brac University

Co-Supervisor:
(Member)

Fakhruddin Gazzali
Lecturer
Department of Computer Science and Engineering
Brac University

Program Coordinator:
(Member)

Md. Golam Rabiul Alam, PhD
Professor
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Sadia Hamid Kazi, PhD
Chairperson and Associate Professor
Department of Computer Science and Engineering
Brac University

Abstract

The reliability of software systems lies in effective test case design, especially in mutation testing, where the objective is to detect subtle faults introduced as mutants. Manually writing unit tests is time-consuming and increases development costs. While traditional test generation tools like Pynguin and Klara provide baseline automation, recent advancements in Large Language Models (LLMs) offer a promising alternative. In this research, we first conducted a comparative analysis between four state-of-the-art LLMs: ChatGPT, Gemini, Claude, and Llama and traditional test generation tools for Python, using mutation score as the primary metric. Motivated by the superior performance of LLMs, we developed a fully automated test generation tool that leverages prompt-based LLMs to create unit tests for Python functions. This tool not only generates syntactically valid and executable tests but also evaluates their quality using both mutation score (via MutPy) and test coverage metrics. The tool employs prompt-engineering strategies, repair loops, and error-driven feedback to iteratively refine failing test cases. The results show that our tool can be reliably harnessed for automated test generation and can outperform traditional tools in both mutation detection and coverage on Python programs.

Keywords: Software Testing; Test Case generation; Prompt-Engineering; LLM-generated Test Cases; Mutation Testing

Table of Contents

Chapter 1

Introduction

1.1 Background

During the 1980s, software issues in the Therac-25, a computerized radiation therapy machine, led several patients to receive excessive radiation, causing several injuries and even death [1]. Software development is an extensive process carried out in multiple systematic steps, and software testing is one of the most crucial steps in this development process. About 30 to 50 percent of software development is estimated to entail software testing [2]. According to the book [3], "Testing is the process of executing a program with the intent of finding errors." If we trace back to the history of software testing, it was first considered a matter of imagination, not automation, and as a fundamentally "unalgorithmic" process [4]. However, over the years, we came to the understanding that software testing is a complex process that needs attention to improve software development.

Software testing has improved drastically over time. Here in this paper, some forms of software testing have been studied carefully to understand the depth of the field and its prospects. The basic form of software testing is unit testing, where emphasis is placed on a specific unit or component of a code. Unit testing is a popular approach, where programmers write test code along with the program code to detect bugs early in the development process [5]. However, this method is not reliable in finding all the bugs present in the entire system during code compilation. Whereas, Regression testing is mainly implemented for continuous integration (CI). Regression testing is a testing process that is applied after a program is modified [6]. This is done after the primary development stage of the software till its full-fledged maturity.

On the other hand, Mutation Testing, which is the primary focus of our paper, works by introducing syntactic changes to the program. Each modified version of the code is called a mutant, and a mutation test intends to generate a set of test cases that can meet the adequacy criterion to successfully detect all possible mutants [7]. A mutation score is calculated by the total number of killed mutants over the total number of mutants, and this score is used to assess the quality of test cases. A major part of Mutation Testing is generating quality test cases capable of detecting a majority of the injected mutants. In a perfect world, developers would want to fully test every possible order of the program; however, that is not possible or feasible, as an infinite amount of test cases would be required for that

[8]. So, software developers can manually code test cases for their programs, but that can be time-consuming and introduce human error, thus increasing the cost of developing the software. On the other hand, automated Test Case Generation (TCG) can suffer from limited test suite coverage. Over the years, multiple methods have been proposed to design test cases, and we reviewed these methods in our paper. Our research takes a hybrid approach, combining LLM-generated test cases and comparing them with traditional methods. However, executing all the test cases demands development time and manual intervention to fix errors. To alleviate this, we extended our work to automate the whole process of generating and repairing buggy tests through a self-healing loop.

1.2 Problem Statement

Though Mutation testing has its usefulness in the software testing field, the full potential of such testing has not been achieved yet. There are some tradeoffs between efficiency, cost reduction, time consumption, and quality of generated test cases. As a result of such problems, software development faces many challenges such as inadequate budget crises, low-quality software in the market, lack of confidence among the developers as well as the consumers, etc.

One of the key steps in Mutation Testing is the process of generating test cases. For software developers, time and money are both crucial and limited resources, so researchers have found numerous ways to automate test case generation to save developers valuable resources. Our literature review explores the traditional methods of automatic test case generation, such as random, path-oriented, goal-oriented, search-based approaches, and more. However, these methods still have some limitations, for instance, the test case might not satisfy all requirements, it might not work with complex applications or generate tests with assertations that cannot properly validate the behavior of the Program Under Test (PUT) [7], [9], [10]. Furthermore, even when high-quality tests are generated, they create a significant maintenance burden. When these tests fail—either due to their own brittleness or changes in the source code—they require manual debugging and repair, re-introducing the high cost and time consumption that automation aims to solve. This room for improvement leaves us with a chance to work in the field, and we aim to integrate new technologies such as Large Language Models to design, evaluate, and automatically repair test cases for Mutation Testing.

1.3 Research Objectives

This research focuses on two main objectives: generating effective test cases for mutation testing with Large Language Models and using traditional methods, and then developing an LLM-based framework for automated test case generation and mutation testing evaluation.

- Explore different Large Language Models in generating test cases.
- Use various Machine Learning techniques during prompting in LLMs.

- Compare the effectiveness of LLM-generated vs traditional test case generation methods.
- Develop a novel framework for the automated repair of failing test cases using an LLM-powered, self-healing loop.
- Evaluate the quality of the final, repaired test suites using comprehensive metrics, including mutation and coverage scores.

Chapter 2

Literature Review

One of the major problems with mutation testing is the sheer number of mutants that need to be executed even for a small program, making Mutation Testing (MT) costly in terms of time and computational power. The paper [11] tries to shrink the large number of mutants that have to be evaluated and consequently assess the quality of test suits without needing to test them against all possible mutants. Joanna Strug and Barbara Strug [11] represented each mutant as graphs to calculate a similarity score among programs, and this is used with a machine-learning (ML) classification algorithm to predict whether a test case is likely to detect the mutant or not. This ML approach did show some encouraging results but also has some concerns. The authors got the best correct classification score (accuracy in predicting where a TS correctly classifies mutants) with the HCFG kernel-based approach, with the highest being 85.1% for a particular test suit. However, some particular test suites showed poorer accuracy, which could be due to randomly partitioning the set of mutants for training. Therefore, future work could include a selection of a variety of mutation operators for the training dataset or using other ML algorithms, such as support vector machines.

The authors Haga and Suehiro [12] emphasized increasing the efficiency of unit testing in software mutation analysis by integrating the concept of Genetic Algorithm. The concept proposed by the paper focused on the derivation of test cases. At first random test cases are created to check the mutation score. If the mutation score is sufficient then GA(Genetic Algorithm) is ignored and the result is final. Redundant test cases are removed using the Detection Matrix. But when the mutation score is not sufficient then GA is applied to refine the test cases by executing a crossover operation increasing the number of test cases by n!, for n elements in one test case. From the paper [12], the average mutation score using GA is 0.979 (137 sec) and without GA is 0.966 (101 sec) with the theoretical upper limit of the score being 0.988.

The paper [13] highlights the difficulty of evaluating test cases and identifies software mutation testing as a solution to determine the effectiveness of the test cases. However, mutation testing produces a large number of mutants, increasing computational cost. So the paper proposes a method to reduce the number of mutants and at the same time create higher-quality test cases. Du et al. [13] proposed a mutation operator selection strategy, where they picked 5 out of the 19 mutation

operators from Mujava to create a subset. Then they combined mutation testing with a genetic algorithm to create higher-quality test cases. After using the selection strategy, the authors saw an average variation score of more than 95% on the variants of the complete set. Even the optimized test case generation method had higher coverage and mutation scores than popular tools like Randoop and Evosuite. The paper focused solely on Java programs, so the method could have been tried on different programming languages or project types for better analysis of its results.

Sharama and Todwal [14] focuses on Regression Testing (Regression Testing is one of the most important aspects related to the software evolution through its various versions over time) to improve the rate of Fault Detection for test case prioritization. The algorithm that has been applied here is the Intelligent Water Drop (IWD) Algorithm. It is a swarm-based approach for Optimal Prioritization. The primary focus of this paper revolves around finding the best possible path for testing through an iterative approach for each test case and adding the finding (The best possible path) to the Total path for every test case leading to the final path being the one and only optimal path. The approach taken by the researchers here is similar to the Knapsack Problem analogy. The problem with this sort of approach is that it cannot distinguish between Equivalent Mutants leaving a margin for error.

Mahdeih et al.[15] emphasizes Coverage Based Test Case Prioritization (TCP) considering fault proneness distribution over code units. The method used in this study depends on collecting the bug history of a particular software to determine a predictive neural network model. This neural network model then estimates the fault proneness for each section of source code incorporating the estimations in coverage-based TCP models. As a method of solving the prioritization problem, the researcher updated existing traditional Additional and Total TCP strategies known as Modified Additional and Modified Total TCP strategies. Experimental observation finds a difference of 40ms in execution time. The empirical study shows an improvement of 4.63% for Modified Additional TCP over Traditional Additional TCP and for Modified Total TCP 3.60% over Traditional Total TCP strategy for 5 projects that were used in this paper, [15].

The study [16] focuses on solving the issues regarding Test Case Generation (TCG) for software testing such as the low efficiency of TCG techniques with limited resources and ignoring requirements processing. Nicha and Jirapun [16] propose a new '2D-5A-4D' process, where there are two processes: Define (1st) and Design (2nd). As highlighted in the literature review section of the paper, requirements prioritization is a critical step in the software testing process, so the new proposed method adds a new requirement prioritization process between the two processes to address the problems of the existing test case generation methods. The result shows that the method presented in the paper [16] generates the smallest size of test cases, at 80.80% whereas the other techniques exceed 97%, with maximum critical domain coverage of 53.20%, which is more than double the other methods, and the fasted TCG process compared to three other test case generation techniques used as benchmarks. However, the experiment to evaluate the methods was done with small-scale data, so results with a larger dataset need to be evaluated to further confirm the effectiveness of the proposed method.

Automated test generation tools exist to ease the process of developers in testing their software. The paper [9] addresses the challenge of generating tests with not only good coverage but also high efficiency in revealing bugs. The authors introduced a new method MuTAP (Mutation Test case generation using Augmented Prompt) to improve bug detection capabilities in test cases. MuTAP works by using zero-shot and few-shot learning with LLMs to generate an initial test case, which is then refined, meaning any faults in the code of the generated test cases are resolved before conducting Mutation Testing. If there are any surviving mutants in the initial test case, then it passes this mutant information to augment the initial prompt and iteratively generate improved test cases until all mutants have been killed or no improvements can be made. MuTAP is successful in outperforming both Pynguin, an advanced test generation tool, and traditional LLM-based approaches. Using llama-2-chat as its LLMC with the few-shot prompt, MuTAP achieves a Mutation Score (MS) of 93.57% and generates test cases with MS=100% for up to 70% of PUTs (Programs Under Test), whereas Pynguin got 65.94% Mutation Score, and only 28.22% PUTs with MS=100%, highlighting the effectiveness of this proposed technique [9]. As noted by Dakhel et al. [9], their method can be applied to other Python programs and even altered to work with other programming languages; however, different LLMs can be fined-tuned with unique instructions so the results may vary a lot and even repairing syntax errors in test cases can alter the results depending on the severity of the error.

Arasteh and Ghaffari [17] aim to increase the efficiency of mutation testing by isolating and clustering single-line redundant mutants using the slicing technique. They emphasize fault-prone areas of the code to expect redundant mutants. Then these areas are sliced up and processed using the parser to generate string-based filtering instructions. Depending on the instructions the redundant mutants are clustered into groups with the same functionality. As a result, when one mutant is killed from one group other redundant mutants are also neutralized, improving the efficiency and reducing the time for software testing. Empirical Study shows that the results of experiments on the Java benchmarks indicate that the proposed method causes a 53.51% reduction in the number of mutants and a 57.64% time reduction compared to similar experiments in the MuJava and MuClipse tools found from the paper [17].

Automated Test Case Generation (TCG) can significantly reduce time and cost for developers, but current algorithms still lack in producing highly organized input data and preserving patterns in test models. Olsthoorn et al. [18] try to solve this problem by using various parts of AI. The author proposed combining evolutionary algorithms (EAs) with grammar-based fuzzing to make the input easier to analyze and interpret. The proposed method [18] was implemented in EVOSUITE, a TCG tool for JAVA, and it achieved a +50% (compared to EVOSUITE without fuzzing) branch coverage in one of the benchmarks. Next, to preserve patterns of method sequences in TCG, the author used EAs with ML to gain additional information about the design and connection between actions within test cases. This info was then used to increase efficiency by prioritizing actions that fit these patterns. This proposed approach named LT-MOSA was compared with two highly developed algorithms, Many Independent Objective (MIO) and Many Objective Search Algorithm

(MOSA). LT-MOSA outperformed both MIO and MOSA in test coverage, achieving 11.7% and 8.5% higher coverage, respectively [18]. The authors also focused on other research areas, such as creating complex objects during TCG from Behavioral Models, Improving Search Guidance for Explicit Contracts, and more. In the future, the author will use the above research areas to create a production-level testing framework.

Khan et al. [19] emphasize the essential of software reliability. Acceptance, Black Box, White Box, Compatibility, Functional, Integration, and Unit are different types of software tests. Genetic algorithms mimic evolution and genetics to improve results. They cover Unit Testing and Genetic Algorithms. Tests are randomized, if tests pass, the process stops. After injecting a mutant it calculates a mutation score. Software testing stops if it fails and poor tests improve with scores. Test genetic algorithms again. Changing mutation and combining crossover improves testing. The algorithm checks if x or y is 1 and then calculates using a loop. Mutating the algorithm, such as changing symbols, to see how it affects the results. This procedure covered 20 mutant suits 100% by the authors [19]. Transforming input data to binary digits, modifying one-bit permutation, and identifying two crossing locations covered all mutants. Mutation analysis and genetic algorithms improve autonomous test circumstances. They enhanced mutation scores over randomly generated test cases with mutation and crossover genetic algorithms. The new method enhances software testing.

Another paper from Khan et al.[20] explain how autonomous test case generation cuts software development costs. Software testing generates and completes test cases automatically. Hybrid Genetic Algorithms fix software bugs for smooth operation. The development of software involves problem testing. Mutation testing and path testing, which check all programming pathways, are crucial. It ensures software functions. The C# tools create graphs. This program prints i if j is greater than half of i. A CFG shows code variables and program flow. Mutants invaded the prime generators. First loop increment, second loop comparison, modifying the condition in the first if statement, and adjusting the condition in the second if statement. Changes can affect program execution and prime number production. A simple C program detects primes between two given numbers. Loops and conditionals check prime numbers. Get input, calculate primes, and show results. Hybrid Genetic Algorithms generate software test cases automatically. Software application test scenarios are conveniently created using this method. They complete all paths or stop after 1000 tries. They innovate through selection, crossover, and mutation. The fitness function compares new and old methods. Mutating the software checks error detection. It considers how well the code covers different paths or outcomes to compute mutation scores. It assesses program flexibility and flaws. They tested computer software test case creation approaches. Testing random, genetic algorithm (GA), and hybrid genetic algorithm (HGA) for greatest mutation score. Genetic algorithms designed mutation test scenarios for this study. Hybrid genetic algorithms produced the best mutation scores. The proposed method yielded the best computer software change test cases.

From [21], Mutants are prioritized and developed through programming. Focusing

on important mutations can identify more programming issues, saving time and money. This ensures that the software works and is tested. Create and test mutants to identify computer code problems. This simulates real-world failures and finds software bugs, enhancing testing. Different methods like mutation testing, generate test data quickly to detect and fix software faults. Weak mutation testing changes code to assess software robustness. Prioritizing computer code versions by relationship. Knowing which versions matter enables them to test critical software early. Find and test important program components to find bugs early. To improve data quality, important parts are checked first. They describe code component prioritization and test data creation. These strategies should help them find and fix program errors quickly. In experiments, researchers test their method. They ask questions, build programs, sort mutant branches, and evaluate results. To detect software vulnerabilities, researchers classify mutant branches by dominance. Researchers assess software error testing. They identify software bugs in tests using weak and strong mutation testing. Researchers examine data output efficiency and software problem detection. Check computer program errors. MuClipse Java mutation testing changes code to fix bugs. Software defect detection test data generation is studied. The report claims DS detects software faults better than PS and RS. The DS method prioritizes feature-based test data to detect bugs faster. Focusing on vital areas first reduces test objects, speeding their technique. Prioritizing these places speeds up error detection and correction, improving testing. Program error detection requires test data. Early testing looks good, but they need to test more to make sure their strategy works for all programs. Mutation testing detects software bugs. Ordering mutant branches boosts test efficiency and validity. Testing should minimize the burden and uncover as many faults as feasible.

For effective tests, object-oriented mutation testing creates programming errors. Mutation testing and genetic algorithms automate test case production and reduce evolutionary mutation testing computation expenses. Researchers improve test case generation and problem identification with object states and control flow. Enhancing genetic algorithms' fitness functions helps them find appropriate mutation testing methods. This paper [22] presents an evolutionary mutation testing fitness function for object-oriented programming that uses object state and control flow information to assess test case fitness. Mutated programs with distinct outputs on different execution pathways have logical faults that the fitness function finds. Genetic algorithms generate test cases for evolutionary mutation testing, speeding up testing. Tests remove non-equivalent mutations to boost program mutation scores. Different fitness functions for evolutionary mutation testing of software program mutant killing test cases. These fitness functions improve mutation testing by assessing reachability, necessity, sufficiency, and mutation impact. Create Java unit testing test cases in EvoSuite using mutation operators and other test data creation methods. Object state fitness checks if a test case is in the right state, whereas control flow information calculates sufficiency cost by detecting logical programming errors. SbRC, Necessity Cost, and COSC assess test case fitness. Fitness tests Java upgrades in eMuJava. The software lets researchers test Java programming capabilities and fitness. To evaluate a fitness function, start experiments and analyze with eMuJava. A computer system comprising hardware and software was used for investigations. Experimental genetic algorithm limitations were examined later. The

paper describes random testing and genetic algorithms for computer software experimentation. Experiments targeted programming shortcomings for improvement. A novel fitness function method detected programming problems. Genetic algorithms compare EvoSuite with eMuJava. Their improved genetic algorithm in eMuJava eradicated computer programming mutants faster than EvoSuite. Software bugs might be hard to identify, lowering performance. Evolutionary mutation testing improves software testing. It detects program errors using state and behaviour data.

Pan et al. [23] emphasized the methods, features, evaluation techniques, and comparison between different ML-based TSP approaches. The author discussed the above concepts by answering 5 research questions. RQ1 (Research Question) tries to find suitable approaches for TSP using ML. The methods the author focused on in this paper are the RL-based approach, Unsupervised Learning, Supervised Learning, and NLP-based TSP techniques. The conclusion for RQ1 unlike RL-based approaches UL and SL-based approaches cannot be used for CI (Continues Integration), further research is needed for the NLP-based method. For RQ2 the author studied different features that are used by ML-based TSP techniques. The features described here are code complexity, Textual data, Coverage information, Execution history, and User inputs. In response to the above question, further research should be conducted focusing on CI techniques elevating the efficiency of old features and integrating new features. RQ3 discussed the 8 metrics for ML-based TSP evaluation which were categorized into 2 sections, Specific Evaluation Metrics for TSP (APFD and its extensions) and General Evaluation Metrics for TSP (Accuracy, Precision, Recall F1-score, NRPA). RQ4 studies performance comparison between ML-based TSP techniques. An overview of the previously used strategies and their effectiveness can be understood from the data presented in this paper for the above question. For RQ5 the previous papers are compared based on Reproducible and Repeatable. The paper [23] analyzed the previous papers to set a table to answer the above question based on the working of earlier papers.

Chen et al. [24] focused on automated unit test generation through the development of an LLM-based tool named *ChatUniTest*. This tool employs an adaptive focal context generation mechanism after scanning each Java file and converting it into an Abstract Syntax Tree (AST). The tool utilizes a **Chain-of-Thought prompting strategy** to guide the LLM in reasoning through code structure and test case generation. Code Llama served as the underlying LLM for ChatUniTest. The system includes three types of validation: Syntactic Validation, Compile-time Validation, and Runtime Validation. ChatUniTest achieved a line coverage of **59.6%**, outperforming both EvoSuite and TestSpark.

For generating unit tests using Generative AI, Bhatia et al. [25] proposed a technique that combines LLMs like ChatGPT with traditional tools like Pynguin by leveraging the minimal overlap between the test cases generated by each, aiming to enhance overall test performance. The dataset was categorised into three types of code units: procedural scripts, function-based code, and class-based code. The evaluation metrics included statement coverage, branch coverage, and correctness of the generated test cases. The study also found that while LLMs can improve coverage, they tend to introduce a higher rate of assertion errors: an issue not commonly

observed in traditional test generation methods.

Chen et al. [26] present a vision paper that reframes prompt construction for LLM applications as a first-class software-engineering problem—what they call "promptware engineering." The authors first contrast traditional software with promptware, noting two fundamental differences: (i) prompts rely on ambiguous, context-dependent natural language rather than formal programming languages, and (ii) LLMs act as probabilistic, non-deterministic runtime environments rather than deterministic execution engines. Building on these insights, the paper proposes a systematic engineering roadmap encompassing requirements, design, implementation, testing/debugging, and evolution. For each phase the authors outline concrete research opportunities (O1–O24)—for example, LLM-driven requirements analysis, formalized prompt design patterns, prompt-centric IDEs, metamorphic testing for prompts, and specialized version-control mechanisms. This roadmap adapts decades of software-engineering best practices to the unique challenges of prompt development, aiming to move practitioners away from today's ad-hoc "prompt hacking" toward disciplined, repeatable processes. Overall, Chen et al. shift the conversation from isolated prompt-engineering tips to a holistic, life-cycle–oriented framework, positioning promptware engineering as an emerging research frontier for the software-engineering community.

Sahoo et al. [27] provides a comprehensive overview of prompt engineering, highlighting it as a crucial method for expanding the capabilities of Large Language Models (LLMs) and Vision-Language Models (VLMs) without altering their core parameters. The authors categorize over 29 distinct prompt engineering techniques, detailing their methodologies, applications, the models involved, and the datasets utilized. The survey explores a wide range of techniques, from foundational methods like zero-shot and few-shot prompting to advanced approaches such as Chain-of-Thought (CoT) prompting for complex reasoning and Retrieval Augmented Generation (RAG) for reducing hallucinations. By systematically analyzing these diverse methods, the paper aims to bridge existing gaps in the literature, offering insights into their strengths, limitations, and potential for future research, while also addressing ethical considerations in the field.

Chapter 3

Preliminary Analysis

3.1 Description of Models

Large Language Models (LLMs) have grown tremendously over recent years and are now widely used across multiple industries. For our study, we picked 4 renowned Large Language Models: Chatgpt 4-o, Gemini 1.5 Flash, Claude 3.5 Sonnet, and Llama 3.1.

All of these LLMs are trained on huge datasets, consisting of texts from books, online forums, code repositories, and many other sources. The essential phases of a transformer-based large language model begins with Data Preprocessing, which involves cleaning, noise reduction, and integration of incoming text from several sources to assure quality data. The text is then tokenized, which involves turning words into token IDs and embedding them in high-dimensional vectors so that the model comprehends semantic meaning. During Pre-training, the model learns language structures and patterns from massive data sets, and Parameter Tuning modifies model weights to increase accuracy.[28]

Transformers use a combination of encoders, decoders, and attention mechanisms. The encoder processes the input data and transforms it into a contextualised representation, while the decoder generates the output sequence based on this representation. The attention mechanism plays an important role by allowing the model to focus on different parts of the input sequence selectively. Thus, it enhances the transformer's ability to capture relationships and dependencies between tokens.

Our main focus was using these LLMs to generate software test cases for a piece of code and run mutation tests to evaluate the quality of these generated test cases using mutation scores. We then pitted these LLM-generated test cases against two existing automated test case generation tools for Python: Pynguin and Klara.

PYNGUIN is a state-of-the-art automated test generation framework for Python. It supports multiple algorithms for its test case generation, but we used DYNAMOSA, which is a search-based evolutionary algorithm, as it is the best-performing algorithm for PYNGUIN [29].

Klara, on the other hand, is an automated test case generation tool for Python. It

operates at the Abstract Syntax Tree (AST), allowing it to analyse the structure of code without running it. This approach makes it much safer to generate test cases, as there are no side effects such as your files being deleted or modified after running the code. While Klara is not considered state-of-the-art, it serves as a useful benchmark for evaluating LLM-generated test quality.

3.2 Dataset

This research used a dataset known as HumanEval specifically designed to evaluate the performance of code-generation models. From the 164 Python programming files, we randomly picked 20 of them, ranging from easy to medium levels. Each file consisted of a Python function that was used as the program under test (PUT), meaning test cases were generated for that Python code using LLMs and automated tools of PYNGUIN and Klara.

In addition, we selected seven additional Python functions from real-world programs to diversify the evaluation dataset. This introduces a more challenging evaluation scenario for the code generation tools, enabling a more comprehensive assessment of each model's performance.

3.3 Comparison Result

For our testing, we had to ensure that the same prompt was used to generate test cases for each of the LLMs. We passed the initial prompt: *"Generate test case for the following code for mutation testing"* and added the Program Under Test (PUT) alongside it in the same input field. We then took the generated test cases from the four LLMs and test cases from PYNGUIN and Klara then used MutPy to generate mutants for the PUT and calculate the Mutation Score (MS). We used the Mutation Score to compare the different models, as MS is a good metric to evaluate the fault-finding capabilities of a test case [29].

Figure 3.1: Mutation Score of all models

For better comparison, we took the average of all the MS found from the 27 PUT, and another average MS from 20 PUT (excluding the 7 real-world programs).

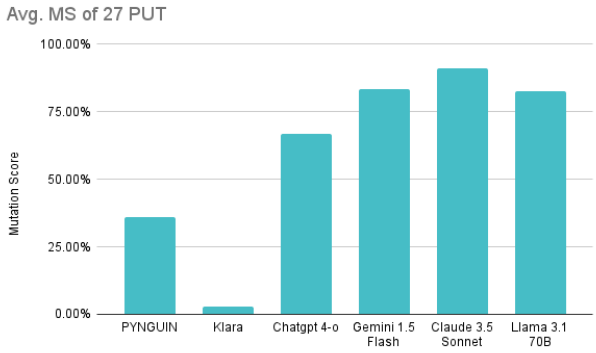


Figure 3.2: Average MS of 27 PUT

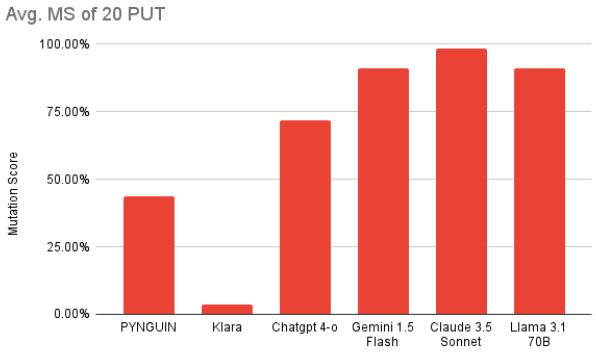


Figure 3.3: Average MS of 20 PUT

$$\text{Actual Mutation Score} = \left(\frac{\text{No. of killed mutants}}{\text{Total no. of mutants}}\right) \times 100$$

The mutation score calculated by MutPy excludes mutants that are incompetant or timed-out. So the MutPy mutation score is calculated in this manner, MutPy mutation score = No. of killed mutants/(total - incompetent - timed-out) mutants.

	PYNGUIN			Klara			Llama 3.1 70B			Chatgpt 4-o			Gemini 1.5 Flash			Claude 3.5 Sonnet		
Operations	Killed	Total	%kill	Killed	Total	%kill	Killed	Total	%kill	Killed	Total	%kill	Killed	Total	%kill	Killed	Total	%kill
AOR	8	43	18.60%	0	39	0.00%	40	48	83.33%	28	42	66.67%	30	42	71.43%	37	42	88.10%
COI	8	22	36.36%	1	24	4.17%	21	21	100.00%	18	26	69.23%	23	24	95.83%	24	24	100.00%
ROR	8	29	27.59%	0	27	0.00%	23	27	85.19%	19	29	65.52%	21	28	75.00%	26	28	92.86%
ASR	1	13	7.69%	0	13	0.00%	12	15	80.00%	7	14	50.00%	11	14	78.57%	12	14	85.71%
AOD	3	8	37.50%	0	8	0.00%	7	8	87.50%	5	8	62.50%	6	8	75.00%	8	8	100.00%
COD	3	5	60.00%	0	4	0.00%	3	3	100.00%	3	5	60.00%	4	4	100.00%	4	4	100.00%
SIR	5	8	62.50%	0	6	0.00%	12	12	100.00%	3	8	37.50%	7	7	100.00%	7	7	100.00%
EHD	1	1	100.00%	0	1	0.00%	1	1	100.00%	1	1	100.00%	1	1	100.00%	1	1	100.00%
LCR	0	2	0.00%	0	2	0.00%	2	2	100.00%	0	2	0.00%	2	2	100.00%	2	2	100.00%
CRP	0	0	Null	0	0	Null	74	92	80.43%	0	0	Null	0	0	Null	50	50	100.00%

Table 3.1: Operators wise comparison of Mutation Testing Results for Various Models

To obtain the actual MS, we identified all the generated mutants and checked if the respective test cases of each model killed them. This approach yields a score that accounts for mutants leading to incompetent or timed-out results.

PYNGUIN: For 20 codes from the HumanEval data set, we generated 131 mutants. Of these, 37 were killed, leading to an actual Mutation Score (MS) of 28.24%. However, the average MS from MutPy is 43.83%, more than the actual MS we calculated manually. The reasoning is that MutPy leaves out only the Survived mutants, leaving a margin for error as the incompetent and timed-out mutants are cancelled from the total mutants calculation.

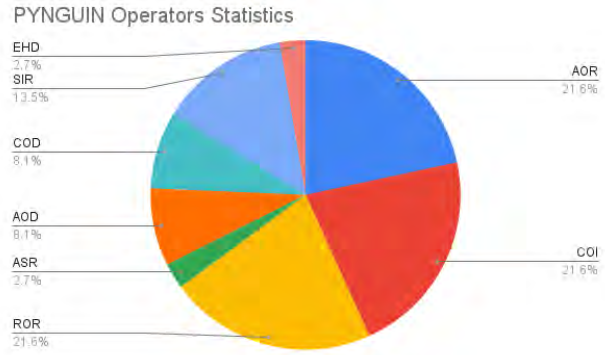


Figure 3.4: PYNGUIN Operators wise mutants distribution

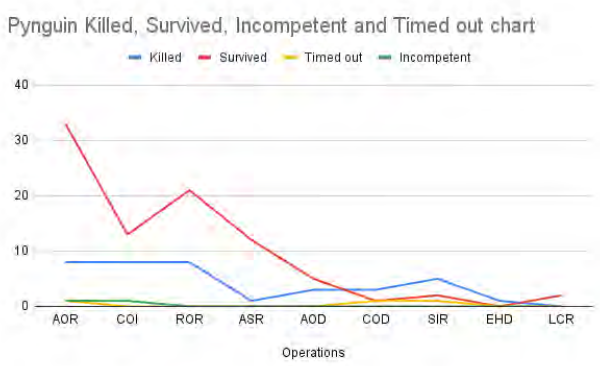


Figure 3.5: PYNGUIN mutants status

Klara: For the same 20 codes from the HumanEval data set, we generated 124 mutants. Among these only 1 mutant was killed by Klara, so the actual MS is 0.806%. The average MS from MutPy is 4.03%. The same reasoning is also implemented here as the MutPy MS is higher than the actual MS.

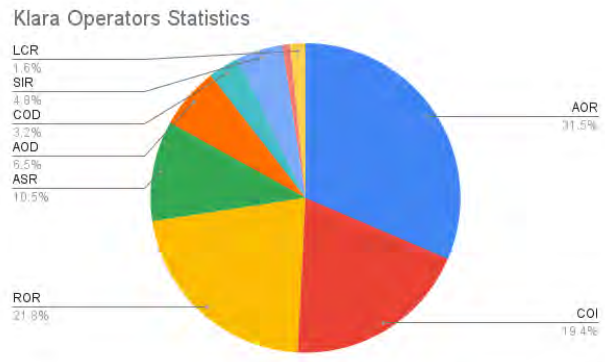


Figure 3.6: Klara Operators wise mutants distribution

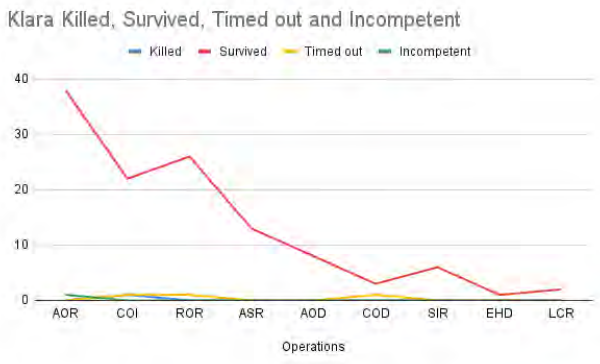


Figure 3.7: Klara mutants status

Llama 3.1 70B: For the same set of codes, the total number of mutants was 229, and the killed mutants were 195. The actual MS is 85.15%. The average MS is 91.21%.

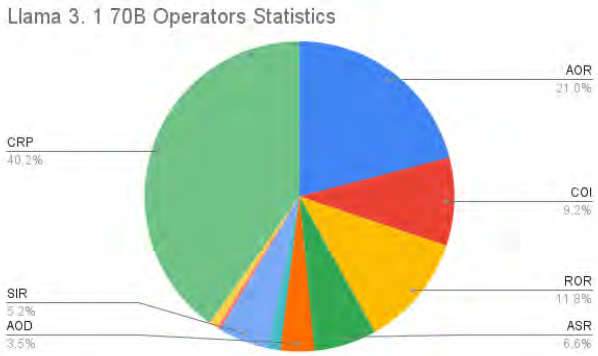


Figure 3.8: Llama 3.1 70B Operators wise mutants distribution

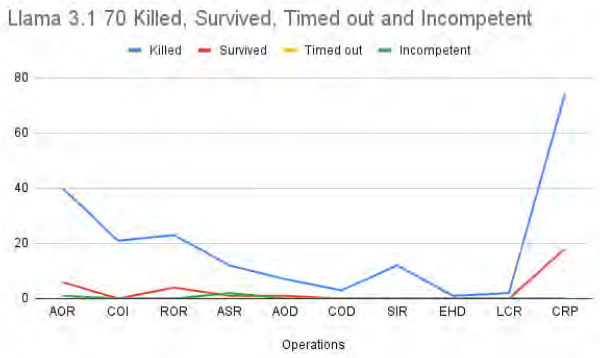


Figure 3.9: Llama 3.1 70B mutants status

ChatGpt 4-o: For ChatGpt, the total number of mutants generated was 135, and the killed mutants were 84. The actual MS is 62.22%. The average MS is 71.75%.

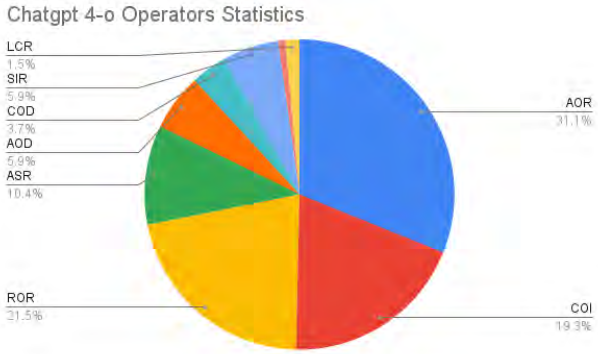


Figure 3.10: ChatGpt 4-o Operators wise mutants distribution

Figure 3.11: ChatGpt 4-o mutants status

Gemini 1.5 Flash: For Gemini, the total number of mutants generated was 130, and the killed mutants were 105. The actual MS is 80.76%. The average MS is 91.10%.

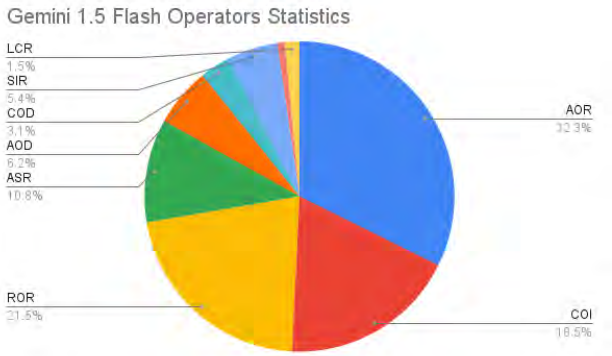


Figure 3.12: Gemini 1.5 Flash Operators wise mutants distribution

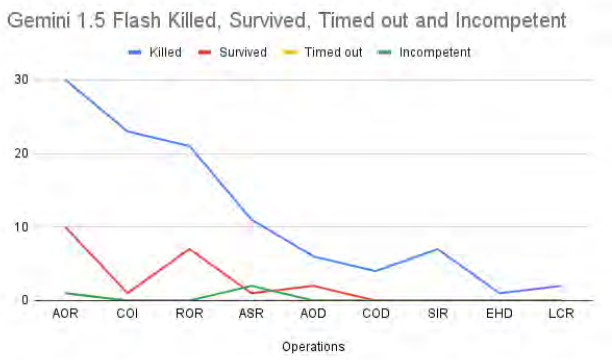


Figure 3.13: Gemini 1.5 Flash mutants status

Claude 3.5 Sonnet: For Claude, the total number of mutants generated was 180, and the killed mutants were 171. The actual MS is 95%. The average MS is 98.00%.

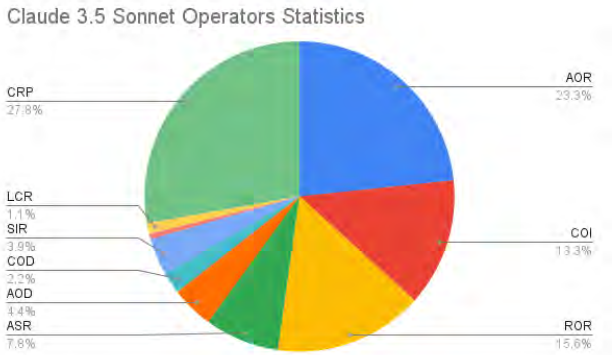


Figure 3.14: Claude 3.5 Sonnet Operators wise mutants distribution

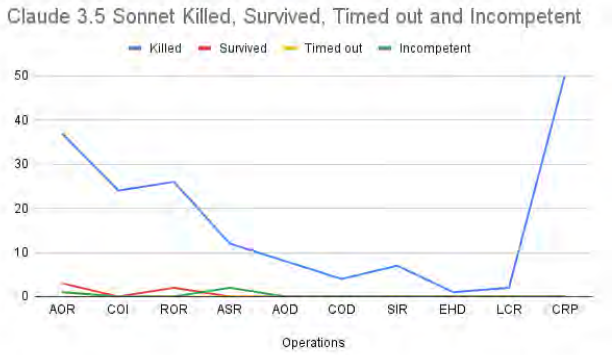


Figure 3.15: Claude 3.5 Sonnet mutants status

From the above data, it is clear that all the LLM-based models provide better test cases than traditional test case generation tools like PYNGUIN and Klara. However, the catch is that the test cases provided by LLM need refining through prompting. Based on the findings we can conclude that for the 20 codes from the HumanEval data set Claude provided the best test cases and the worst results came from Klara.

The poor results of Klara indicate that generating test cases by working on the AST level and not executing the code leads to low-quality test cases.

Whereas, LLMs performed exceptionally as anticipated, except for a slightly lower average MS of ChatGPT 4-o. This could be due to how the LLMs were trained and how they are tuned behind the scenes.

Chapter 4

Methodology

4.1 Workflow

Observing the exceptional results of the LLMs, we proceeded to develop an automated framework for generating and evaluating unit tests for Python programs. While in our comparison, Claude performed the best in terms of Mutation Score, we decided to go for Google's Gemini LLM using their API. Google's Gemini is widely recognised and used in this field and has a free tier for research purposes.

Our automated framework **ASTRA** [30] (Automated Software Testing, Repair, and Analysis) provides an end-to-end, automated solution for generating and evaluating Python unit tests.

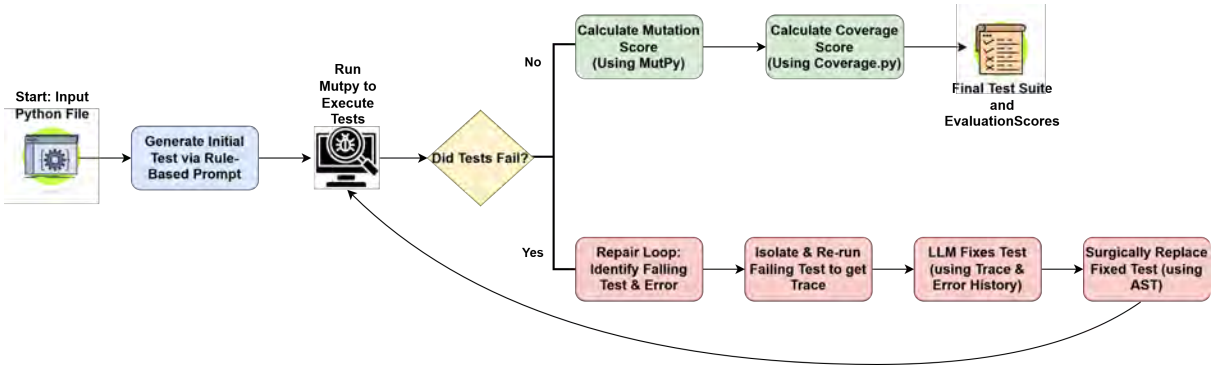


Figure 4.1: Workflow of ASTRA

The workflow can be broken down into three primary stages: initial test generation, an iterative test repair cycle, and final quality evaluation.

1) **Initial test generation:** ASTRA starts by taking in a Python file as the input source code. The whole file is read and passed to the LLM alongside a carefully designed prompt that guides the model to produce complete, runnable tests adhering closely to the source code's behavior. At this stage, a baseline test suite is generated with multiple assertions to cover edge cases and is ready for immediate execution and evaluation.

2) **Iterative test repair cycle:** the initially generated test cases now enter a cycle of test execution, failure detection, and automated repair. The generated test suite is executed against the source code using MutPy. Upon encountering any test failures, the system performs a detailed failure analysis by isolating the failing test and capturing its error traceback along with dynamic execution traces. This contextual information is then used to invoke the LLM again, this time through another specialized prompt designed to encourage stepwise reasoning and precise correction of the faulty test code.

The framework applies a targeted code replacement strategy to update only the specific failing test function, preserving the remainder of the test suite. This surgical modification is performed using abstract syntax tree (AST) parsing and rewriting to ensure structural integrity. After patching, the revised test suite is re-executed to validate the repair. The cycle of execution, analysis, and repair repeats until all tests pass or a predefined iteration limit is reached. To mitigate repetitive failures and encourage novel solutions, the framework incrementally increases the LLM's randomness parameter (temperature) with each subsequent repair attempt on the same test, promoting diversity in generated fixes.

3) **Final quality evaluation:** Once a fully passing test suite is obtained, the framework conducts a comprehensive quality assessment based on two standard metrics: mutation score and code coverage. The mutation score, provided by MutPy, reflects the proportion of artificially introduced faults that the test suite successfully detects, serving as an indicator of fault-detection capability. The coverage metric measures the extent to which the test suite exercises the source code's lines and branches, thereby assessing its thoroughness. The final deliverable consists of an automated, validated test suite along with quantitative metrics that display its quality and completeness.

4.2 Prompt Engineering

The effectiveness of ASTRA's framework relies heavily on the design of our carefully engineered prompts for the LLM. Throughout development, our prompting strategies evolved through three distinct phases—each addressing specific limitations of its predecessor. This iterative refinement culminated in a hybrid prompt design that balances strict constraint adherence with contextual depth and analytical reasoning. Our approach to prompt engineering aligns with the growing understanding of its critical role in leveraging Large Language Models, as extensively surveyed in recent literature [27]. We categorise our prompting strategies into two primary tasks: test generation and test repair, each employing different techniques tailored to the nature of the task.

Phase 1: Zero-Shot Prompting (Baseline)

In the initial phase, prompts were minimal, typically instructing the LLM to "analyse the source code and generate test cases." This zero-shot approach lacked structural guidance or contextual grounding. As a result, it often led to several issues. The LLM often produced hallucinated fixes, attempting to "correct" perceived bugs in the source code and generating tests that failed against the actual implementation. Additionally, the output quality was inconsistent; the generated tests frequently missed edge cases, lacked coherence, and did not follow established best practices. The repair process was also ineffective, as it tended to enter loops of superficial fixes due to the LLM not having enough contextual information about the failure it was trying to address.

Phase 2: Rule-Based and Instruction-Following Prompts

To overcome these limitations, we adopted a rule-based prompting strategy for test generation. This approach imposed a set of explicit, non-negotiable instructions that constrained the model's creativity and focused its output on the precise task at hand. Key rules included:

- The source code is the single source of truth.
- Tests must reflect the actual behaviour, even if the code contains bugs.
- No assumptions should be made about the expected output; assertions must be based solely on the source code's behaviour.

These constraints transformed the LLM's role from an autonomous code writer into a verifiable test generator, significantly improving the correctness, robustness, and structure of the output.

Phase 3: Chain-of-Thought Reasoning with Contextual Injection

While the rule-based approach sufficed for generating initial tests, repairing failing tests required a more sophisticated prompt design. We adopted a hybrid technique for test repair that integrated:

- **Chain-of-Thought (CoT)** Reasoning: The repair prompt required the LLM to perform step-by-step analytical reasoning before proposing a fix. The structure explicitly instructed the model to:

  (A) Identify the failing test

  (B) Analyse the actual versus expected behaviour

  (C) Apply the fix accordingly

  This integration of CoT reasoning is a well-established method for enhancing LLM performance on complex tasks requiring logical progression [27].

- **Rich Context Injection:** In addition to the source code and the full test suite, the prompt dynamically included the exact pytest error traceback, a dynamic execution trace generated by rerunning the failing test in isolation, a log of previous failed repair attempts, annotated with explicit warnings to avoid repeating the same fixes.

This contextual depth enabled the LLM to act more like a debugging assistant, grounded in actual execution behaviour rather than assumptions, leading to more accurate and sustainable repairs.

- **In-Context Learning from Failure:** To prevent repetitive repair loops, the prompt explicitly informed the LLM of past failed attempts. This self-corrective mechanism discouraged redundant suggestions and encouraged novel solutions. Additionally, we gradually increased the LLM's temperature during repeated repair attempts, introducing more variability to escape local optima in the solution space.

Chapter 5

Evaluation and Result

5.1 Dataset

To evaluate our tool, we used the same HumanEval dataset, but for a more comprehensive evaluation, we took 70 Python functions. Both MutPy and Pynguin have limitations when dealing with native extensions (e.g., C-based modules) or Python files with complex external dependencies. So we had to selectively choose Python modules that did not include such libraries or dependencies.

We also picked 18 Python projects, previously used in a similar study by Bhatia, to compare our results and ensure a diversified dataset. These projects contain Function-based modular code, exhibiting minimal encapsulation and include dependency on global variables.

5.2 Analysis

Each of these files was used as the input source code for ASTRA, and it generated a test suite, fixed errors, if present, and even gave us metrics such as Mutation Score (using MutPy) and Coverage metrics for easier evaluation.

To ensure a rigorous evaluation of the generated test suites, our framework measures code coverage using the coverage.py library with branch coverage enabled (–branch). Unlike standard line coverage, which only confirms whether a line of code has been executed, branch coverage provides a stricter assessment by verifying that every possible outcome of each decision point (e.g., if/else statements) has been tested.

This is a crucial distinction, as high line coverage can be misleading if important logical paths remain untested. By using branch coverage as one of our key quality metrics, we ensure that the test suites generated by our LLM-powered framework are not only executing the code but are also thoroughly validating its conditional logic, leading to a more reliable and meaningful assessment of test quality.

Metric	Average Value
Number of Test Cases	18.47
Mutation Score (%)	96.01
Coverage (%)	98.64

Table 5.1: Average test statistics for 70 Python functions from the HumanEval dataset

MuTAP, proposed by Dakhel et al. [9], was evaluated on 164 Python programs from the HumanEval benchmark to assess the bug detection capabilities of LLM-generated test cases using mutation testing. In our evaluation, we randomly selected a representative subset of 70 programs from the same dataset. Our tool generated an average of **18.47 test cases** per Program Under Test (PUT), compared to MuTAP's reported average of **2.6 test cases** using its best-performing configuration (few-shot prompt with `llama-2-chat` as the LLM Component). A higher number of test cases does not always mean better test quality or effectiveness. Many test cases may include redundant or shallow tests that inflate coverage but do not improve fault detection.

While our test suites achieved a **code coverage of 98.64%**, it is important to note that coverage alone does not provide a complete picture of test effectiveness. High coverage can still miss critical bugs if the assertions are weak or absent. Therefore, mutation score offers a more reliable measure of a test suite's fault-detection capability.

In this regard, our approach achieved a Mutation Score (MS) of **96.00%**, surpassing MuTAP's best reported MS of 93.57% [9]. Additionally, while MuTAP improves its tests through iterative prompt augmentation using surviving mutants, our method produces highly effective test cases without requiring such post-generation refinement. This reduces both the architectural complexity and the runtime cost of inference.

These results demonstrate the strength of our prompt engineering strategy in generating not only larger but also more effective test suites. It also opens promising directions for future work that may combine our generation mechanism with mutation-aware refinement strategies like those used in MuTAP.

Metric	Average Value
Number of Test Cases	28.06
Mutation Score (%)	92.43
Coverage (%)	96.33

Table 5.2: Average test statistics for 18 Python project files

Bhatia et al. [25] evaluated the coverage achieved by ChatGPT and Pynguin across various Python programs. In the subset of 18 Python files we also tested (drawn from their dataset), they report an average statement and branch coverage of **93.26%**

and **91.68%** for ChatGPT, and **90.3%** and **90.1%** for Pynguin, respectively. Averaging both statement and branch coverage gives an overall mean of **92.47%** for ChatGPT and **90.20%** for Pynguin on these samples.

Our approach achieved a significantly higher average combined coverage (statement + branch) of **96.33%** on the same 18 files. Although coverage is an important metric, it can be misleading on its own, as high coverage does not necessarily imply effective testing unless assertions are meaningful. Therefore, we also report a high mutation score of **96.00%**, supporting the effectiveness of our generated tests in fault detection, not just execution path coverage.

Chapter 6

Limitations and Future Work

While this research demonstrates a viable framework for the automated generation and repair of unit tests, several limitations should be acknowledged. The system's primary dependency on a proprietary, cloud-based LLM like Google Gemini introduces challenges related to data privacy, and the long-term reproducibility of results Consequently, the prompt engineering at the core of our methodology is highly tuned to this specific model, and its effectiveness may not directly transfer to other LLMs without significant re-tuning. Furthermore, the current strategy of including the entire codebase as context presents scalability issues for very large projects, which may exceed the model's context window. The framework also currently lacks automated mechanisms for mocking complex external dependencies such as databases or network services, limiting its applicability on highly integrated, enterprise-level codebases.

The limitations of the current framework highlight several key directions for future research. A primary objective is to enhance the system's accessibility and modularity by integrating support for other LLMs or even open-source LLMs that can be run locally, thereby addressing privacy concerns and removing operational costs. In parallel, we plan to refactor the architecture to be language-agnostic, as the core structure of ASTRA is designed to work with any programming language. This involves creating a pluggable system architecture that can be easily extended to support additional programming languages—such as JavaScript by implementing language-specific adapters for test execution and code parsing.

Chapter 7

Conclusion

In this paper, we addressed the significant challenge of creating and maintaining robust unit test suites in modern software development. Our preliminary analysis confirmed that state-of-the-art Large Language Models substantially outperform traditional test generation tools like Pynguin and Klara, which motivated the development of our novel framework, ASTRA. This framework introduces an end-to-end, automated methodology that leverages an LLM for the initial generation, execution, and—most critically—the automated repair of unit tests. By employing a self-healing loop that uses dynamic execution traces and surgical code replacement, ASTRA can autonomously resolve test failures, significantly reducing the need for manual intervention.

Our empirical evaluation demonstrates the remarkable effectiveness of this approach. As shown in Table 5.1, ASTRA achieved an exceptional average mutation score of 96.01% and an average coverage score of 98.64% on the challenging HumanEval dataset. These strong results were further validated on a diverse set of real-world projects, where ASTRA maintained a high average mutation score of 92.43% and coverage of 96.33%. These quantitative results prove that our framework not only generates comprehensive test suites but also ensures they are effective at detecting faults. By successfully automating the entire test lifecycle, ASTRA presents a significant step forward, offering an economically viable and highly efficient solution that has the potential to fundamentally improve the practice of software quality assurance.

Bibliography

[1] N. Leveson and C. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993. DOI: 10.1109/MC.1993.274940.

[2] R. Blanco, J. Tuya, and B. Adenso-Díaz, "Automated test data generation using a scatter search approach," *Information and Software Technology*, vol. 51, pp. 708–720, Apr. 2009. DOI: 10.1016/j.infsof.2008.11.001.

[3] G. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing* (ITPro collection). Wiley, 2011, ISBN: 9781118133156. [Online]. Available: https://books.google.com.bd/books?id=GjyEFPkMCwcC.

[4] H. Leeds and G. Weinberg, *Computer Programming Fundamentals* (La Livre de Poche). McGraw-Hill, 1966. [Online]. Available: https://books.google.com.bd/books?id=5ZdQAAAAMAAJ.

[5] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014, pp. 201–211. DOI: 10.1109/ISSRE.2014.11.

[6] H. Leung and L. White, "Insights into regression testing (software testing)," in *Proceedings. Conference on Software Maintenance - 1989*, 1989, pp. 60–69. DOI: 10.1109/ICSM.1989.65194.

[7] B. Falah and S. Hamimoune, "Mutation testing techniques: A comparative study," Nov. 2016. DOI: 10.1109/ICEMIS.2016.7745368.

[8] M. Keyvanpour, H. Homayouni, and H. Shirazi, "Automatic software test case generation: An analytical classification framework," *International Journal of Software Engineering and its Applications*, vol. 6, pp. 1–16, Jan. 2012.

[9] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, "Effective test generation using pre-trained large language models and mutation testing," *Information and Software Technology*, vol. 171, p. 107 468, 2024, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2024.107468. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584924000739.

[10] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, "Revisiting test smells in automatically generated tests: Limitations, pitfalls, and opportunities," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 523–533. DOI: 10.1109/ICSME46990.2020.00056.

[11] J. Strug and B. Strug, "Machine learning approach in mutation testing," in *Testing Software and Systems*, B. Nielsen and C. Weise, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 200–214, ISBN: 978-3-642-34691-0.

[12] H. Haga and A. Suehiro, "Automatic test case generation based on genetic algorithm and mutation analysis," in *2012 IEEE International Conference on Control System, Computing and Engineering*, IEEE, 2012, pp. 119–123.

[13] Y. Du, Y. Pan, H. Ao, N. Ottinah Alexander, and Y. Fan, "Automatic test case generation and optimization based on mutation testing," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2019, pp. 522–523. DOI: 10.1109/QRS-C.2019.00105.

[14] P. Sharma and V. Todwal, "Test case prioritization through efficient mutation analysis using water droplet algorithm," *IJIRST–International Journal for Innovative Research in Science & Technology*, vol. 1, no. 10,

[15] M. Mahdieh, S.-H. Mirian-Hosseinabadi, K. Etemadi, A. Nosrati, and S. Jalali, "Incorporating fault-proneness estimations into coverage-based test case prioritization methods," *Information and Software Technology*, vol. 121, p. 106 269, 2020.

[16] N. Kosindrdec and J. Daengdej, "A test case generation process and technique," *Journal of Software Engineering*, vol. 4, pp. 265–287, Apr. 2010. DOI: 10.3923/jse.2010.265.287.

[17] B. Arasteh and A. Ghaffari, "Efficient software mutation test by clustering the single-line redundant mutants," *Data Technologies and Applications*, 2024.

[18] M. Olsthoorn, "More effective test case generation with multiple tribes of ai," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022, pp. 286–290. DOI: 10.1145/3510454.3517066.

[19] R. Khan and M. Amjad, "Automatic test case generation for unit software testing using genetic algorithm and mutation analysis," in *2015 IEEE UP Section Conference on Electrical Computer and Electronics (UPCON)*, 2015, pp. 1–5. DOI: 10.1109/UPCON.2015.7456734.

[20] R. Khan, M. Amjad, and A. K. Srivastava, "Generation of automatic test cases with mutation analysis and hybrid genetic algorithm," in *2017 3rd International Conference on Computational Intelligence Communication Technology (CICT)*, 2017, pp. 1–4. DOI: 10.1109/CIACT.2017.7977265.

[21] X. Yao, G. Zhang, F. Pan, D.-w. Gong, and C. Wei, "Orderly generation of test data via sorting mutant branches based on their dominance degrees for weak mutation testing," *IEEE Transactions on Software Engineering*, vol. 48, pp. 1169–1184, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:226462137.

[22] M. B. Bashir and A. Nadeem, "Improved genetic algorithm to reduce mutation testing cost," *IEEE Access*, vol. 5, pp. 3657–3674, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:13206446.

[23] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: A systematic literature review," *Empirical Software Engineering*, vol. 27, no. 2, p. 29, 2022.

[24] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, *Chatunitest: A framework for llm-based test generation*, 2024. arXiv: 2305.04764 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2305.04764.

[25] S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote, *Unit test generation using generative ai : A comparative performance analysis of autogeneration tools*, 2024. arXiv: 2312.10622 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2312.10622.

[26] Z. Chen, C. Wang, W. Sun, *et al.*, *Promptware engineering: Software engineering for llm prompt development*, Mar. 2025. DOI: 10.48550/arXiv.2503.02400.

[27] P. Sahoo, A. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, *A systematic survey of prompt engineering in large language models: Techniques and applications*, Feb. 2024. DOI: 10.13140/RG.2.2.13032.65286.

[28] M. A. K. Raiaan, M. S. H. Mukta, K. Fatema, *et al.*, "A review on large language models: Architectures, applications, taxonomies, open issues and challenges," *IEEE Access*, vol. 12, pp. 26 839–26 874, 2024. DOI: 10.1109/ACCESS.2024.3365742.

[29] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 168–172, ISBN: 9781450392235. DOI: 10.1145/3510454.3516829. [Online]. Available: https://doi.org/10.1145/3510454.3516829.

[30] *ASTRA: An automated framework for the generation, repair, and quality evaluation of python unit tests*, https://github.com/SNaf1/ASTRA.