Nile University

School of Information Technology and Computer Science

Program of Artificial Intelligence & Computer Science

AutoTestLoc

AIS496& CSCI496 Senior Project II

Submitted in Partial Fulfilment of the Requirements

For the bachelor's degree in information technology and computer science

Artificial Intelligence & Computer Science

Submitted by

Amira Ahmed Elsharaby 202000674

Doaa Hamed Abd El-Fattah 202002179

Nouran Ahmed Mahmoud 202000574

Rawan Hisham Neiazy 202001119

Yasmeen Elsayed Ismail 211001051

Supervised by

Dr. Tamer Khidr Mohamed Arafa

Giza – Egypt

Spring 2024

Project Summary

Traditional manual testing is known for its sluggish pace and high costs, often resulting in incomplete coverage and overlooked bugs. Studies show that manual testing efforts can uncover only around 80% of defects. Our project, AutoTestLoc, aims to revolutionize this outdated approach by leveraging large language models (LLMs) for automated test case generation and code fault localization. By harnessing the potent capabilities of the Llama2 and Llama3 models, we seek to significantly enhance the effectiveness of testing endeavors for developers. The project addresses the challenges faced in achieving comprehensive test coverage and efficient bug detection by automating the test case generation process and precisely pinpointing the location of problems in the source code. This dual focus on automated test case generation and code fault localization aims to streamline the testing procedure, reduce debugging time and effort, and ultimately enhance the reliability and functionality of software systems. The project addresses the challenges faced in achieving comprehensive test coverage and efficient bug detection by automating the test case generation process and precisely pinpointing the location of problems in the source code. This dual focus on automated test case generation and code fault localization aims to streamline the testing procedure, reduce debugging time and effort, and ultimately enhance the reliability and functionality of software systems. Our goals include speeding up the software testing process through automated test case generation and improving code fault localization by accurately identifying the exact lines and types of errors. AutoTestLoc represents a significant advancement in software quality assurance, offering a cost-effective and dependable solution for improving software reliability and customer satisfaction. Through innovative testing methodologies and the integration of LLM technology, we aim to set new standards in software testing practices and contribute to the ongoing evolution of software development processes.

Keywords:

Automated test case generation, Code fault localization, large language models (LLMs), Llama model, Testing efficiency, Bug detection, Software reliability, Software functionality, Optimization, Debugging, Software Prompt feedback, Issue detection, Defect resolution, Software quality assurance, Cost-effectiveness, Customer satisfaction.

Table of Contents

List of Figures

Chapter 1:

Introduction

1.1 Background:

Software development requires quality assurance because it ensures the dependability and functioning of software products. Testing processes have significantly changed, moving from manual test case execution to programmed, data-automated approaches. It used to take a lot of time to perform manual testing, which involved selecting test cases, executing tests, recording results, and drawing conclusions. Automation has totally changed the conditions under which testing takes place, enabling a variety of data inputs and guaranteeing repeatability and uniformity in the process. Despite these advancements, it is still challenging to acquire objective and comprehensive test coverage, especially when dealing with novel scenarios. The need for a substantial quantity of manual labor is one of the remaining barriers to attaining quality assurance. However, there are still challenges to attaining quality assurance. For example, scripting, testing, and associated operations still involve a large amount of manual effort.

In this context, the use of automated test cases and code fault localization becomes crucial. The depth and breadth of testing are improved by the wide range of scenarios that are covered by automated test case development. This procedure drastically cuts down on the amount of time needed to construct comprehensive test suites and minimizes human error. On the other side, code fault localization is an essential method for locating the precise places in the code where errors happen. By offering accurate defect information, this technique improves debugging and saves time and effort in resolving problems. Software becomes more dependable and resilient when automated test case creation and code defect localization are combined to enhance the testing and debugging process.

By addressing these problems, the project hopes to increase testing environments' precision and effectiveness, which will promote the ongoing development of software quality assurance. The project's importance lies in its capacity to maximize manual labor reduction, streamline testing processes, and ensure software systems' resistance to ever-changing complexities. The effort offers a holistic approach to improving software quality by integrating code fault localization with automated test case development. This will ensure that systems are properly tested and that any faults are promptly and correctly discovered and fixed.

1.2 Motivation:

Automated test case generation and code fault localization holds significant importance in the software development and testing process. Code fault localization is an essential component of the software development lifecycle that helps with successful regression testing, cost savings, more user satisfaction, decreased downtime, better productivity, and efficient debugging. It is essential to keep a robust and stable software system. Moreover, automating testing plays a vital role in managing the complexity of current codebases, ensuring software quality, and avoiding bugs. In addition, manually creating test cases can be time-consuming, arduous, and susceptible to human error. It's crucial to verify that modifications to a program do not negatively impact its original functionality. Developers typically write tests to uncover unintended bugs caused by changes, but not all developers have access to high-quality tests. Additionally, anticipating future changes often requires the creation of new tests. Automated test case generation offers unparalleled efficiency, enabling the creation of a comprehensive suite of test cases rapidly. Integrating automated test case generation and code fault localization improves software development by seamlessly merging efficient testing operations and continuous integration. This method makes the best use of available resources while giving developers prompt feedback and accurate issue detection for easier troubleshooting. The system ensures complete test coverage, which prevents the introduction of new bugs and contributes to improved product quality and customer happiness. All things considered, it promotes flexibility in response to shifts while upholding an affordable and dependable software development process.

1.3 Objectives:

The AutoTestLoc project aims to leverage large language models to enhance automated test case generation and code fault localization in software development. The specific goals of the project are to streamline the software testing process, reduce debugging time and effort, and ultimately improve the quality and reliability of software systems. By utilizing advanced techniques in automated test case generation, the project seeks to identify specific test cases that trigger faults or defects in the software, enabling developers to pinpoint the root cause of failures efficiently. Additionally, the project aims to enhance code fault localization by accurately

identifying the exact location of defects in the source code, thereby guiding developers to swiftly address and rectify issues.

In a scientifically rigorous manner, the project will evaluate the effectiveness of its approach through comprehensive testing and validation phases. These phases will involve assessing the generated test cases for thoroughness and relevance in detecting software faults. Furthermore, the accuracy and efficiency of code fault localization will be evaluated by comparing the identified defect locations with actual problematic code segments. Through rigorous testing procedures and performance metrics, the project intends to demonstrate the efficacy of large language models in automating test case generation and fault localization.

1.4 Scope:

Large Language Models (LLMs) can be used in a wide range of code testing scenarios. LLMs can be used to examine code snippets in automated test case generation, taking advantage of their natural language processing skills to understand code structures. As a result, test suites can be made more thorough and efficient by automatically generating test cases through Llama by finding probable inputs that cover a variety of paths and situations. Furthermore, LLMs are useful in code fault localization because they are used to evaluate failed test cases. Debugging can be accelerated by LLMs' ability to see patterns and comprehend the context of code execution, which helps them locate possible causes of faults in the code. So, our purpose is to perform both automated test case generation and code fault localization highlights how valuable and adaptable these tools are for improving the efficiency and effectiveness of code testing procedures.

1.5 Significance of the Study:

Significant advancements and trends have been seen in the status of test generation employing generative unit testing and pre-trained large language models (LLMs). To guarantee software quality and dependability, LLMs can be employed in software testing. Common LLM kinds, agile engineering techniques, and LLM input strategies are all covered in this paper. Large-scale datasets can be handled, and natural language processing applications can benefit greatly from the performance of LLM technology. Through the creation of more varied test cases, effective code analysis and debugging, and the creation of remedies for issues, LLMs can enhance software

testing approaches. This raises the software's standard and dependability by enabling faster and more precise problem-solving. But more investigation and innovation are still needed in some areas [5] 1.

It also identifies potential opportunities of using lean test methods (LLMs) for unit test generation, highlighting their benefits such as faster development, improved test coverage, enhanced quality assurance, detection of test smells, and exploration of different context styles. LLMs can generate test cases that validate the correctness of code units, helping developers identify and fix defects early in the development cycle. They also serve as a means of quality assurance, validating the functionality and behavior of software components. The study also examined the presence of test smells, such as duplicated assertions and empty tests, in the generated unit tests, helping developers improve the maintainability and effectiveness of their test suites. These findings contribute to the ongoing research and development of automated software testing techniques [1],[2].

Chapter 2:

Related Work

2.1 Introduction to Literature Review:

The purpose of the literature review for the project "Automated Test Cases and Code Fault Localization using Large Language Models" is to critically examine the body of research that exists at the point of intersection of code fault localization, automated testing, and the use of large language models (LLMs). It seeks to identify the complicated strengths and limitations of LLMs, particularly in improving automation and optimizing fault localization effectiveness in software development. Through an examination of critical insights, methodologies, and best practices, the review aims to strategically mold the project's approach. Moreover, it offers an extensive overview of the state-of-the-art now, highlighting efforts that effectively incorporate LLMs into automated testing and code fault localization procedures. This thorough comprehension directs the project as it navigates the complexities and advances the terrain of automated testing practices in software development.

2.2 Historical Perspective:

Software testing's automated test case generation has seen a dramatic change, moving from laborious manual procedures to complex AI-driven techniques. Manual test case creation and execution required a lot of work and was error-prone at first. While early automated tools concentrated on automating repetitive operations, the development of automated technologies in the late 20th century signaled a move towards efficiency. Test case creation was transformed throughout time by approaches like model-based testing and AI-enhanced techniques like neural networks and genetic algorithms, allowing for more thorough and efficient testing.

Further improvements to automated testing have come from parallel developments in code fault localization. Simple heuristics and manual inspection were used in early fault localization techniques, but machine learning, statistical analysis, and spectrum-based approaches are used in more recent ways to pinpoint fault-prone locations. These days, these developments are essential to pipelines for continuous integration and deployment, with reliable solutions provided by platforms augmented by AI and tools like Selenium. Software testing is a dynamic and ever-evolving subject, and the incorporation of AI and ML in self-healing test automation and enhanced fault localization promises to make it even more dependable and effective in the future.

2.3 Theoretical Framework:

In the domain of Test Case Generation and Code Fault Localization, the project benefits from the integration of various pertinent theories and models, each playing a pivotal role in augmenting our comprehension and enhancing project effectiveness. Notably, the SBST approach serves as a foundational framework for systematically generating test cases, while symbolic execution models, exemplified by concolic testing, provide valuable insights into the exploration of diverse program paths. These theories facilitate systematic code path exploration, identification of potential faults, and improvement in the overall quality of generated test cases [3]. In the early stages, testing automation was done in a semi-computerized manner, using partial automation to reduce manual work but not achieving full automation. Then, specialized tools for automated testing surfaced, empowering developers to create and automate test cases. Model-based testing became popular in the 2000s, involving the building of abstract models for the system under test, from which test cases were automatically generated. Algorithms for search-based testing become more popular in the 2010s.

These techniques automatically generated test cases by methodically exploring a program's input space using search strategies. Later, automated test case generation became a fundamental aspect of the software development process, especially when combined with continuous integration and continuous deployment (CI/CD) pipelines. Advances in static analysis and symbolic execution approaches have helped to generate more accurate test cases and identify issues. Machine learning applications in automated testing leverage algorithms to learn from past results, identify patterns, and optimize test scenario generation, enhancing testing effectiveness [4].

Furthermore, the project draws on the utility of LLMs, exemplified by the BART transformer model and CodeBERT, to enhance the accuracy and efficiency of test case generation. As few-shot learners, these models demonstrate proficiency in synthesizing test cases and effectively localizing faults. Complementary contributions from theories such as mutation testing and neural network rankers enrich the project by enhancing bug detection capabilities and facilitating the generation of meaningful test oracles. The integration of these theories and models represents a concerted effort to advance the field of software testing through innovative methodologies [2].

In the specific context of Test Case Generation and Code Fault Localization, the project derives significant advantages from relevant theories and models, encompassing fault localization techniques, test case generation algorithms, and Large Language Models (LLMs) for code analysis. These frameworks contribute substantively to the project by offering systematic approaches to identify and isolate faults, generate effective test cases, and leverage advanced language models to comprehend code semantics. The integration of these theories aims to enhance the accuracy and efficiency of fault localization processes, elevate the quality of generated test cases, and streamline debugging workflows for software developers [5].

Within the same domain, several pertinent theories and models play a crucial role in advancing our understanding of the project. SBFL and MBFL theories provide structured frameworks for identifying potential fault locations within the codebase. These models significantly contribute to the project by offering systematic approaches to pinpointing bugs and generating test cases that effectively target identified faults. The utilization of these theories establishes a methodological foundation for improving software quality, refining debugging processes, and facilitating more efficient and accurate identification of code faults [6].

Using deep learning techniques, notably Convolutional Neural Networks (CNNs), this system conceptualizes fault localization as an image pattern recognition job that aims to effectively detect defective code segments and derive meaningful representations from code coverage matrices. Code Coverage Representation Learning, a crucial component, converts these matrices into a format that can be fed into CNN. It does this by highlighting patterns that differentiate between defective statements and not, and by combining test case results to identify lines that are prone to errors. The framework improves the efficiency and accuracy of fault localization by prioritizing code segments that appear suspicious based on an analysis of previously observed faults and similarities with current difficulties. Furthermore, by examining data dependencies inside execution flows and data flows, it looks at links between statements to find the effects of flawed code on related statements, even those far from the defective line. By utilizing dynamic code coverage data and static code representation learning techniques, integrating these components into the DEEPRL4FL (Deep Reinforcement Learning for Fault Localization) framework seeks to increase fault localization efficiency and accuracy while offering a reliable method for locating faulty code segments [16].

2.4 Previous Research and Studies:

Upon reviewing the published work “Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction”, existing methods primarily focus on increasing coverage or generating exploratory inputs, often falling short of achieving more semantic objectives such as reproducing bugs specified in bug reports. The limitations of current techniques lie in their exclusive attention to program crashes, neglecting a significant portion of bug reports dealing with semantic issues. However, a notable finding from empirical studies indicates that a considerable number of tests are added due to bug reports, constituting a substantial portion of overall test suite size. Recognizing the underappreciated yet impactful role of generating bug reproducing tests from bug reports, this paper introduces the LIBRO framework. Leveraging Large Language Models (LLMs), LIBRO aims to address the gap in knowledge regarding LLMs' reliability in generating tests from bug reports. Through extensive empirical experiments, the study validates LIBRO's ability to generate bug reproducing tests, offering a promising avenue for significantly enhancing developer efficiency by automating the test generation process from bug reports [7].

The examination of previous research in the domain of "Software Testing with Large Language Models: Survey, Landscape, and Vision" reveals several key findings and notable gaps in existing knowledge. The survey highlights that Large Language Models (LLMs), such as T5 and GPT-3, have emerged as transformative tools in natural language processing and artificial intelligence, exhibiting exceptional performance across various coding-related tasks. However, the survey underscores the challenges persisting in automated unit test case generation, mobile GUI testing, and other software testing tasks, despite the application of diverse approaches. Notably, existing methods, including search-based, constraint-based, and random-based techniques, have limitations in generating satisfactory test suites and achieving comprehensive coverage. The landscape review demonstrates the potential of LLMs, like ChatGPT1 and LLaMA2, with billions of parameters, in addressing complex practical tasks, particularly in code generation and software testing. However, the survey identifies a gap in the understanding of how LLMs can effectively address semantic challenges unique to software testing. The vision presented in the study emphasizes the need for innovative techniques and the integration of LLMs to enhance software testing efficacy. The identified gap lies in the exploration of advanced prompt engineering methods and the incorporation of LLMs with traditional testing techniques. Overall, the synthesis of key

findings and gaps provides a foundation for future research, pointing towards the necessity of refining LLM applications and advancing methodologies to overcome existing challenges in the realm of software testing [1].

Fault localization has been extensively researched in software engineering, focusing on identifying faulty program elements through passing and failing test executions, a process inherently tied to program semantics. Traditional Spectrum-Based Fault Localization (SBFL) methods infer the suspiciousness of program elements based on coverage information but often overlook semantic aspects, leading to inaccuracies. Advanced techniques like Mutation-Based Fault Localization (MBFL) and Angelic Debugging delve deeper into program semantics by generating multiple mutants for each element and using symbolic analysis to correct test outcomes, respectively, but these methods face significant computational challenges. The SmartFL approach aims to balance effectiveness and scalability by modeling only the correctness of program values rather than their full semantics, integrating static analysis and dynamic execution traces to create a probabilistic model. Evaluated on the Defects4J dataset, SmartFL achieved a top-1 statement-level accuracy of 21% and significantly reduced the average time cost per fault to 210 seconds, outperforming state-of-the-art methods. This novel approach improves fault localization by providing a more efficient yet accurate analysis, marking a significant advancement over traditional and existing methods in the field [13]

The authors of the paper "Large Language Models for Test-Free Fault Localization" outlined or contrasted with a few important topics from earlier work. They made use of well-known benchmarks and datasets, such Defects4J, which comes in several versions, most notably V1.2.0, which has 395 problems from 6 projects, and V2.0.0, which has more flaws. These benchmarks made sure that comparisons that are often utilized in fault localization research were standardized. Additionally, the efficacy of the model in identifying security vulnerabilities and working across several languages was assessed using the BugsInPy dataset, which contains 493 issues from 17 distinct Python projects, and the Devign dataset, which has 5,260 defects from two open-source C projects. The study expands on several faulty localization methods. When calculating suspiciousness ratings based on test coverage, while it mainly depends on test coverage to compute suspiciousness ratings, Spectrum-Based Fault Localization (SBFL) may overlook some incorrect

behaviors. By employing mutants to understand the effect of code parts on test outcomes, Mutation-Based Fault Localization (MBFL) improves Support-Based Fault Localization (SBFL). However, MBFL has difficulties when it is unable to instantiate acceptable mutants. Deep learning models trained on data sources like as program structure and test coverage are used by Machine Learning-Based Fault Localization (MLFL) approaches, such DeepFL, DeepRL4FL, and TRANSFER-FL, to forecast defective lines in the code. The work makes use of recent developments in language models for code, such Codex and GPT-Neo, which, although occasionally producing faulty code, have greatly enhanced performance in tasks like code completion and generating code from natural language descriptions [14].

The use of large language models (LLMs) for software engineering activities, such as automatic program repair (APR), has been thoroughly investigated in prior research and studies. According to recent studies, LLMs are quite effective at correcting defects, even when employed without any additional fine-tuning. Unfortunately, these studies frequently don't fully examine issue localization or how it ties in with problem fixes. Empirical research has demonstrated the efficacy of LLMs in code creation, vulnerability discovery, fuzzing, and software specification development. However, these tasks usually need extensive data fine-tuning or the application of quick design approaches. In this work, we explicitly study APR and find that better prompts lead to much better fine-tuning results, which may have applications in other software engineering activities that involve LLMs. Using well-known datasets like CodeXGLUE and CodeReviewer, the bug-fixing framework was evaluated. Further analyses were conducted on the unseen dataset Defects4J in order to reduce any biases and guarantee wider applicability. The study underscores the significance of accurate prompt design and draws attention to the difficulties caused by discrepancies between problem localization and repairing models during LLM fine-tuning. To address these discrepancies and enhance the accuracy of bug-fixing, an intermediary adjustment module is suggested. Toggle, the suggested framework, demonstrated notable gains in bug-fixing accuracy across many benchmark evaluations. It offered innovations including localizing and resolving flaws at token granularity and a new design of various prompts to optimize LLMs as bug-fixing models [15].

2.5 Current State of the Field:

Large language models (LLMs) have revolutionized AI and natural language processing, enhancing test coverage and error detection. T5 and GPT-3 models, pre-trained on large corpora, have shown impressive results in NLP tasks. ChatGPT and LLaMA, with billions of parameters, offer new opportunities for using LLMs in software testing. LLMs are commonly used for creating unit test cases, test oracles, and system test inputs. Recent research suggests that LLMs can provide feedback-directed test generation while maintaining high code coverage. Methods like CODAMOSA and ATHENATEST have utilized LLM-generated test cases to improve search-based testing and achieve higher code coverage [2].

From the challenges that limited capacity to discover bugs Although the use of large language models (LLMs) to generate test cases has been investigated, many of these studies concentrate on code coverage as opposed to issue discovery. It is well known that there is only a modest correlation between test coverage and test effectiveness in identifying bugs and post-processing procedures. Test cases generated by LLMs may have syntactic or functional mistakes, making them unsuitable for testing corner situations in Programs Under Test (PUTs) and identifying specific kinds of problems. Certain post-processing actions are required to improve the efficacy of the test cases that are generated, while the characteristic assertion that unit tests with no assertions or assertions that are too vague are frequently produced by existing automated unit test generation approaches. When evaluating the intended behavior of the PUT, the quality of assertions is a critical factor that requires attention [8],[9].

Large language models (LLMs) for software debugging, especially in fault localization (FL), have made tremendous strides in recent years, but there are still many obstacles to overcome. Promising outcomes have been observed when LLMs are integrated with external tools, improving task performance in areas like defect localization and software repair. Notably, AutoFL a sophisticated FL approach that makes use of LLMs has proven to perform better than more conventional FL techniques like SBFL and MBFL. Metrics like Precision@1, Reciprocal Rank, and Average Precision are used to assess AutoFL's effectiveness and dependability. Furthermore, the accuracy and favorable reception that developers receive from AutoFL's explanations, more than half of which are accurate, are critical factors in the practical deployment of this technology.

There is room for development, though, especially in terms of condensing and improving the usefulness of the explanations. The explanations' thorough nature is appreciated, and developer feedback is largely good. However, recommendations for shortening the explanations and improving their relevance are made. The paper also recognizes that there may be risks to validity, such as computation time fluctuation, the randomness of LLMs, human mistake in assessing explanations, and potential data leaking. The results are primarily restricted to Python and Java projects that use unit tests, and the feedback could not accurately reflect the wide range of software development communities. Despite these difficulties, the use of LLMs in fault localization as demonstrated by programs such as AutoFL represents a potential development in automated debugging; nonetheless, more study is required to overcome existing constraints and improve usefulness [17].

Large language model (LLM) fault localization is a field that has made great strides and is showing promising outcomes right now. Test cases and in-depth program analysis are frequently used in traditional fault localization procedures, which may be time- and labor-intensive. Utilizing LLMs that have been pretrained on large code repositories, like CodeGen, is one of the recent techniques. Explicit test cases are not necessary for these models to comprehend code semantics and detect possible flaws. An LLM-based fault localization tool called LLMAO, for example, has demonstrated its ability to perform better than earlier deep learning-based techniques by successfully localizing security flaws and defects at the line level. This development shows how LLMs have the power to revolutionize fault localization by allowing more effective defect diagnosis and decreasing reliance on labor-intensive human processes [14].

2.6 Test Case Generation Survey:

The graph in Fig 1.that shows the results of a survey question about how people rate the collaboration between developers and testers regarding automated test cases. The survey results show that:

- 15% of respondents rated the collaboration as very good
- 10% of respondents rated the collaboration as good

- 5% of respondents rated the collaboration as excellent
- 65% of respondents rated the collaboration as fair or poor (25% fair, 40% poor)

Overall, it seems that the majority of respondents (65%) believe that the collaboration between developers and testers regarding automated test cases is fair or poor. This suggests that there is room for improvement in this area.

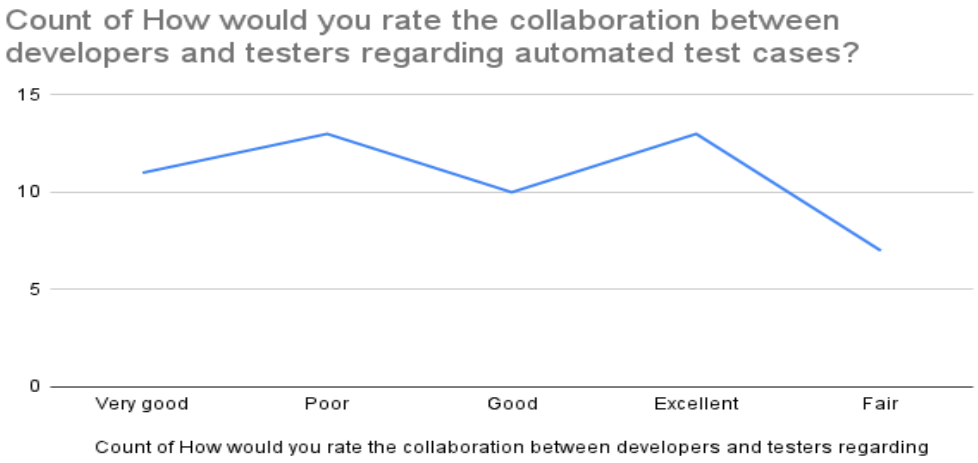

Figure 1

The quantitative data and findings from the pie chart in Fig2.:

- 24.1% of people are very satisfied with the current effectiveness of automated test cases.
- 11.1% of people are satisfied with the current effectiveness of automated test cases.
- 27.8% of people are neutral on the current effectiveness of automated test cases.
- 11.1% of people are dissatisfied with the current effectiveness of automated test cases.
- 25.9% of people are very dissatisfied with the current effectiveness of automated test cases.

Overall, 35.2% of people are satisfied with the current effectiveness of automated test cases, while 64.8% are not satisfied.

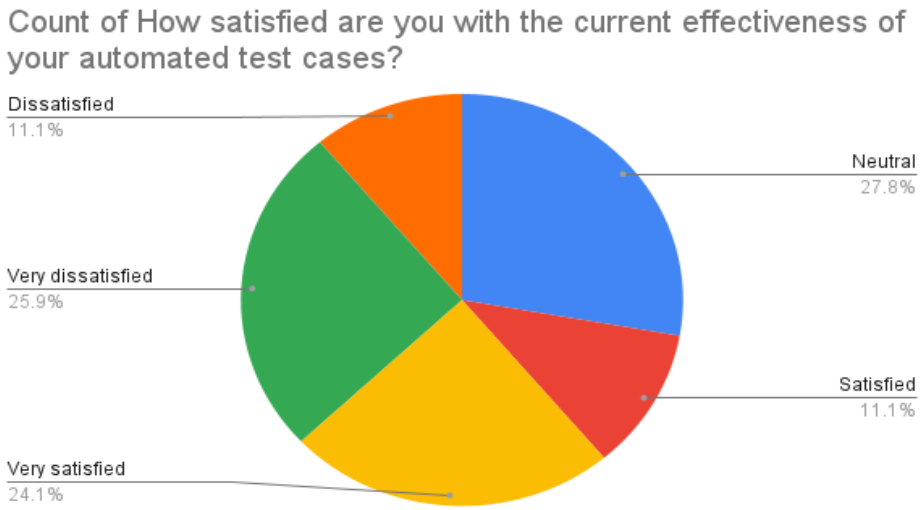

Figure 2

.

The pie chart shows how familiar people are with automated test cases. The slices of the pie chart are labeled with the following answer choices and their corresponding percentages:

- Very familiar: 30.8%
- Extremely familiar: 15.4%
- Moderately familiar: 18.2%
- Somewhat familiar: 17.3%
- Not familiar at all: 17.3%

Overall, 46.2% of people are very or extremely familiar with automated test cases, while 53.8% are somewhat familiar or not familiar at all.

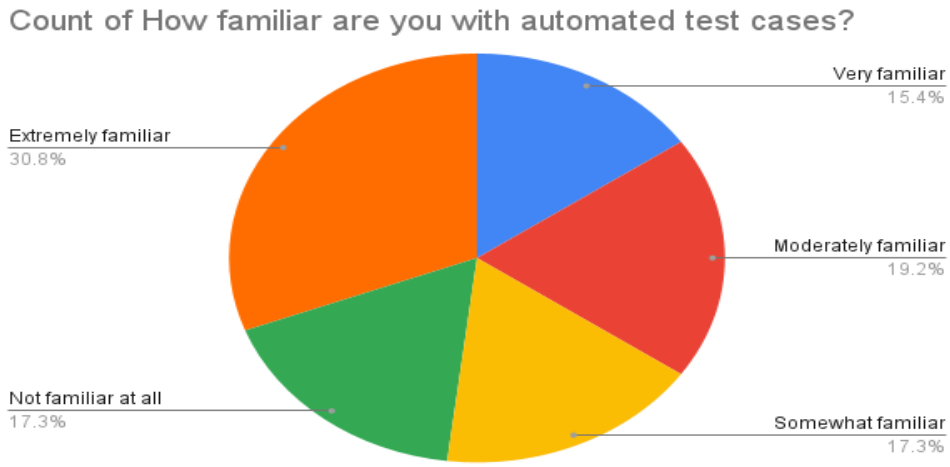

Figure 3

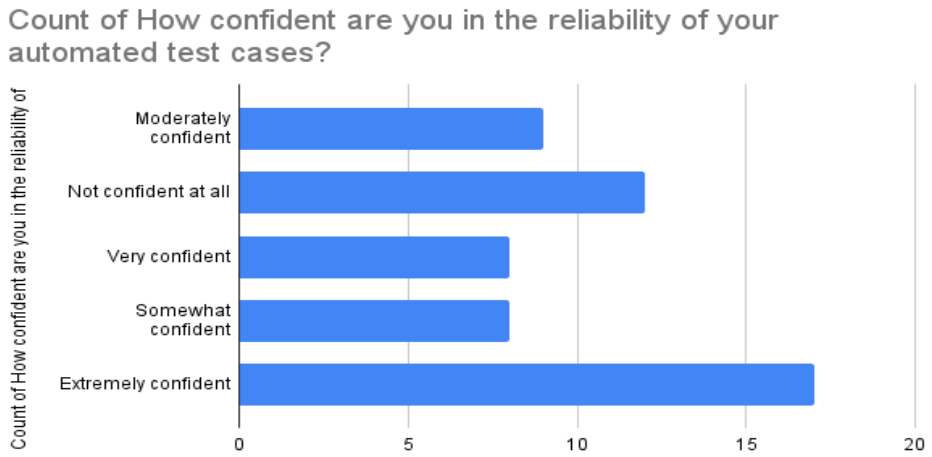

Figure 4

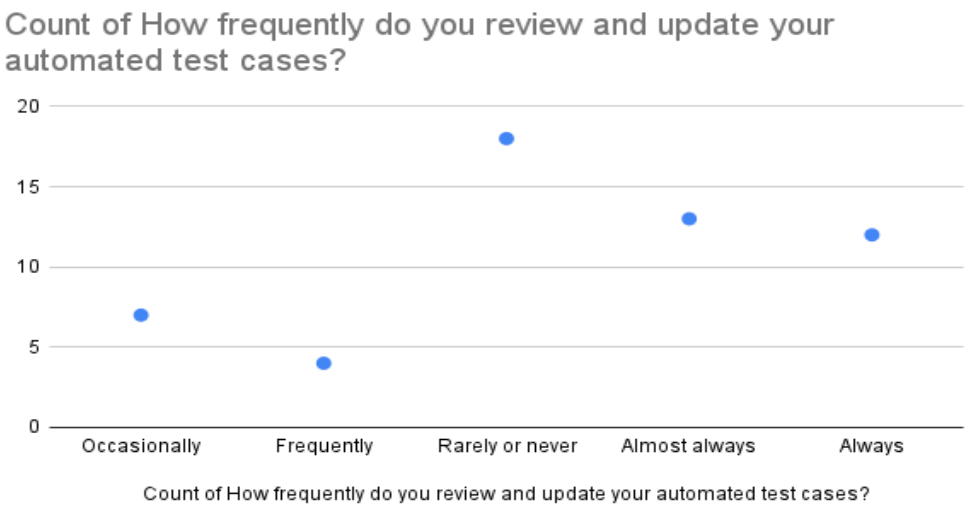

Figure 5

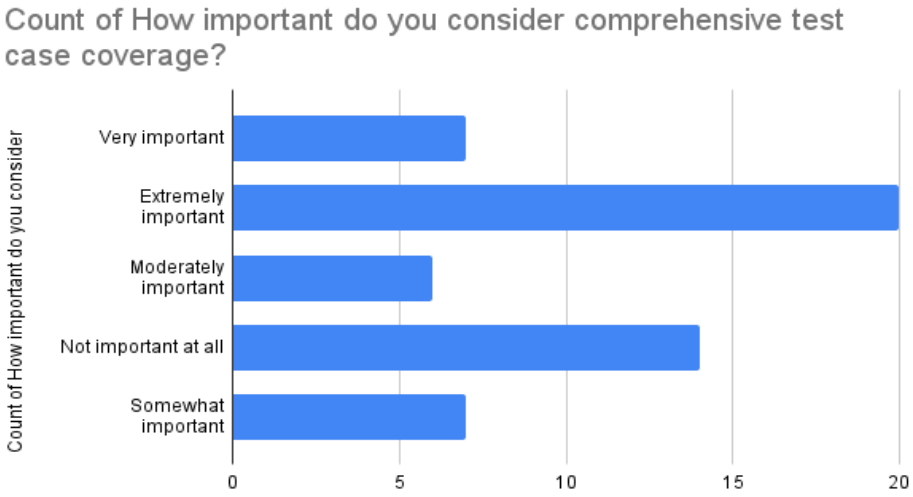

Figure 6

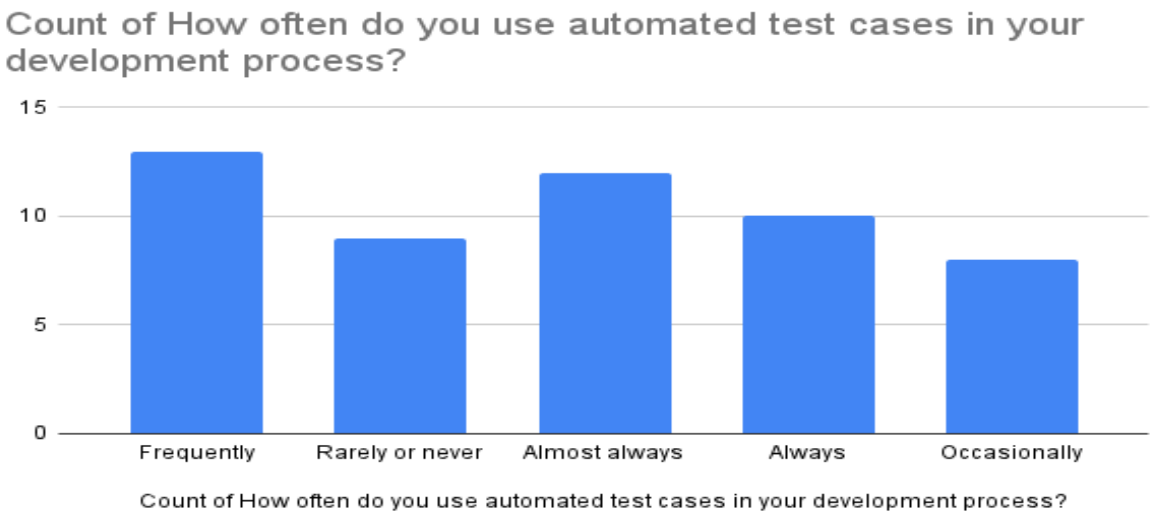
Count of How often do you use automated test cases in your development process?
15
10
5
0
Frequently
Rarely or never
Almost always
Always
Occasionally
Count of How often do you use automated test cases in your development process?

Figure 7

Chapter 3:

Materials and Methods

3.1 System Description:

3.1.1 Automated Test Cases:

Unit test cases for software modules or components are automatically generated by this system using the Llama large language model. By automatically generating test cases based on predefined criteria and the code itself, it seeks to increase the efficacy and efficiency of the software testing process. Using a large language model (LLM) known as Llama, this system seems like an intriguing method for automated software testing. By greatly enhancing efficacy and efficiency, this method has the potential to completely transform software testing. Nonetheless, for the best outcomes, it's critical to understand the constraints and guarantee correct integration. Recall that examining test cases and customizing the system to meet unique requirements still require human skills.

3.1.1.1 Context Functionality:

- **The system takes two main inputs:**
  1- Software parts or modules: Code is written in a variety of programming languages.
  2- Particularised standards: These requirements could include particular user interactions that the test cases should cover, edge cases, fault scenarios, and intended functionalities.

- **Processing:** The LLM, Llama, examines the given code and standards. It makes use of its understanding of software development procedures, natural language, and maybe pre-trained data regarding typical testing patterns to find the pertinent sections of the code: Llama concentrates on portions of code that pertain to the given parameters, including functions, classes, or certain lines of code. Create test cases: Llama uses its knowledge to generate automated test cases that are designed to exercise the code segments that have been found. These examinations may consist of:

  1- verifying function outputs by calling functions with varying inputs.
  2- Configuring parameters to initiate error handling or edge cases.
  3- Testing the behavior of the software by simulating user interactions.

- The output A collection of automatically generated unit test cases that can be run against the software modules is the ultimate result of the unit test case-generating system that uses Llama LLM. Usually, these test cases would be formatted in accordance with the unit testing guidelines of the selected testing framework (e.g., JUnit, NUnit), which would enable easy usage without requiring a lot of manual labor. Count of test cases: Depending on the complexity and the selected testing methodologies, the system may produce more than one test case for every code segment that has been detected. Details of the test case: To check the behavior of the code, each test case should explicitly state the inputs, expected outcomes, and any assertions. Combining testing instruments with integration: The output could be executed and reported on immediately using the current testing tools. User opinions: The system might provide options for users to review and refine the generated test cases before execution.

3.1.1.2 Potential Benefits:

Potential Advantages: Enhanced efficiency: When test cases are generated automatically, developers can save a lot of time and effort by not having to write them by hand.

Enhanced efficacy: Llama may produce more thorough test cases than human techniques by considering a variety of factors and using its knowledge of code. This could result in higher-quality software.

Reduced bias: More objective testing can result from automation's ability to lessen human biases that could affect test case selection.

3.1.1.3 Problems and Things to Think About:

LLM Constraints: The caliber and comprehensiveness of Llama's training data determine how successful the model is, much like any other AI system. Unrepresentative edge situations in its training data, complicated code structures, or testing requirements may cause it to falter.

Test case quality: To guarantee that the created test cases are precise, pertinent, and efficient, human testers may need to examine and improve them.

Integration with current testing procedures: The development process's testing frameworks and tools must be integrated with the system.

For the Unit Test Cases Generation System, you have provided thorough and well-thought-out objectives, requirements, and workable solutions. The essential features required to save human labor, increase code coverage, and give developers and testers efficient testing tools are ranked in order of importance.

3.1.1.4 Strengths:

User-centric perspective: You give priority to solving real-world issues that developers and testers encounter, emphasizing automation, efficacy, and efficiency.

Balanced approach: You recognize the possibility of sophisticated features for future development while also acknowledging the necessity of essential functionalities.

Scalability and flexibility: To ensure broader applicability, you consider the system's capacity to manage a range of project sizes and programming languages.

Integration with current workflows: To encourage easy adoption, you place a strong focus on seamless integration with reporting tools and testing frameworks. Pay attention to support and usability. You recognize the value of an intuitive user interface and easily accessible help files.

3.1.1.5 Addressing the wish list:

Effectiveness metrics: Although it is stated as an objective, it could be useful to include certain metrics, such as defect detection rate or code coverage percentage, to gauge the success of the test cases that are generated.

Security considerations: Security mechanisms ought to be specified as part of the requirements if the system handles sensitive code or data.

Open-source vs. closed-source: To promote community contributions and participation, weigh the possible advantages and disadvantages of opening the system to the public.

Evaluation plan: Establish a strategy for assessing the efficacy and performance of the system, including success criteria and techniques for gathering and analyzing data.

3.1.2 Code Fault Localization:

Code Fault Localization represents a significant leap forward in automated debugging. It leverages the power of Large Language Models (LLMs) from Llama3 to pinpoint code errors with high accuracy. Code Fault Localization goes beyond mere fault localization by offering detailed explanations for each bug, significantly aiding developers in the debugging process. Furthermore, it provides guidance messages to enhance subsequent function call requests, addressing potential shortcomings in user input for debugging functions.

3.1.2.1 Context Functionality and Diagram:

Input: The system takes source code from the developers as input. This can be in various programming languages depending on the LLM's training data.

Processing: The input code is analyzed using the LLM, which understands the code's context and semantics.

Fault Detection: The system compares the code with known fault patterns stored in its database.

Output: The system generates a report indicating the possible locations and nature of faults in the code.

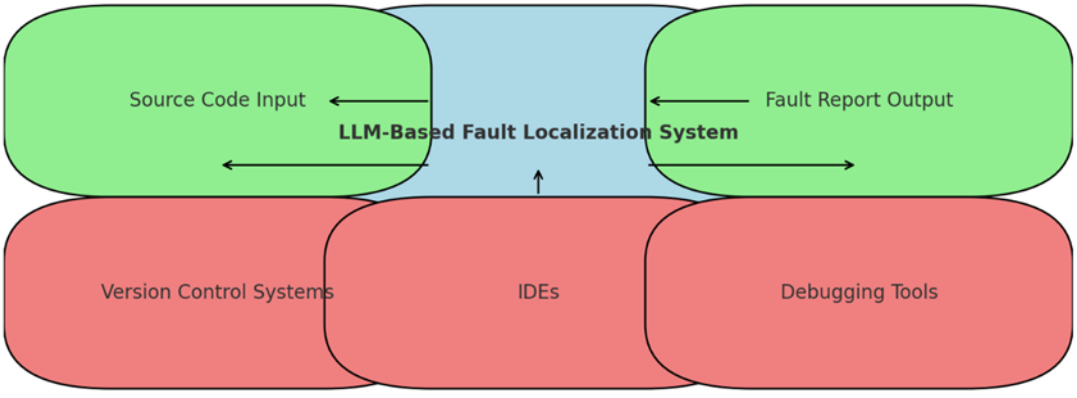

Figure 8: Code Fault Localization context diagram

The detailed context diagram for the LLM-Based Fault Localization System. The diagram illustrates the system, its boundaries, and interactions with external components such as version control systems, integrated development environments (IDEs), and debugging tools.

3.1.2.2 Potential Benefits:

Faster Debugging: Pinpoint issues quickly, letting developers fix them instead of wasting time searching.

High-Precision Fault Detection: Advanced AI ensures accurate bug identification, minimizing missed errors.

Effortless Workflow Integration: Works seamlessly with popular development environments, becoming an intuitive extension of your coding process.

Scales with Your Projects: Handles small projects and massive codebases with ease, making it universally applicable.

Continuously Improves: Learns from user feedback to become a more effective debugging tool over time.

3.1.2.3 Problems and things to think about:

Model Limitations: LLMs, despite their advanced capabilities, can still misinterpret complex code structures or logic, leading to false positives or negatives.

Integration Challenges: Ensuring seamless integration with various IDEs and VCS can be complex due to the diversity of development tools and environments.

Performance Overheads: The fault localization process can be resource-intensive, potentially impacting the system's performance, especially with very large codebases.

Data Privacy: Handling source code, which might include sensitive information, raises concerns about data privacy and security.

User Adoption: Developers might be resistant to adopt new tools, especially if they are comfortable with their existing debugging processes.

The boundaries of the system include the source code input, the fault localization process facilitated by the LLM, and the output which is a report detailing the identified faults. The system does not directly modify the source code but provides recommendations and insights.

3.1.2.4 Strengths:

High Accuracy: Leveraging LLMs for fault localization ensures a high level of accuracy in identifying potential issues within the code. The models are trained on vast amounts of code, enabling them to recognize complex patterns and anomalies that might not be apparent to human developers.

User-Friendly Integration: The tool is designed to integrate seamlessly with popular IDEs and VCS, making it easy for developers to incorporate it into their existing workflows without significant disruptions. This ease of use encourages adoption and regular utilization.

Scalability: The system is built to handle codebases of varying sizes, from small projects to extensive enterprise-level applications. This scalability ensures that it can be a valuable tool for a wide range of development teams.

Continuous Improvement: By incorporating feedback from users, the system continuously learns and improves its fault localization capabilities. This iterative improvement process ensures that the tool remains effective and relevant over time.

3.1.2.5 Addressing the wish list:

Improved Fault Localization Accuracy: Continuous model training and user feedback incorporation can enhance accuracy.

Better Integration with Tools: Regular updates and support for a wide range of development tools ensure seamless integration.

Reduced Performance Impact: Optimization of the algorithm and leveraging efficient computing resources can minimize performance overheads.

Enhanced Privacy Measures: Implementing robust security protocols can address data privacy concerns.

Increased User Adoption: Providing comprehensive documentation, tutorials, and user support can help in driving adoption among developers.

3.2 System Requirements:

1- Automated Test Cases:

The goal of a project involving automated test cases is to expedite and simplify the software testing procedure. The project aims to increase efficiency, detect bugs early in the development cycle, and increase test coverage by automating test case generation, execution, and management. Automation lowers manual labor, speeds up feedback, and improves the overall quality of software. Automated test cases also help with continuous integration and delivery, which guarantees software releases go smoothly and testing procedures are economical. Show (Figure 9)

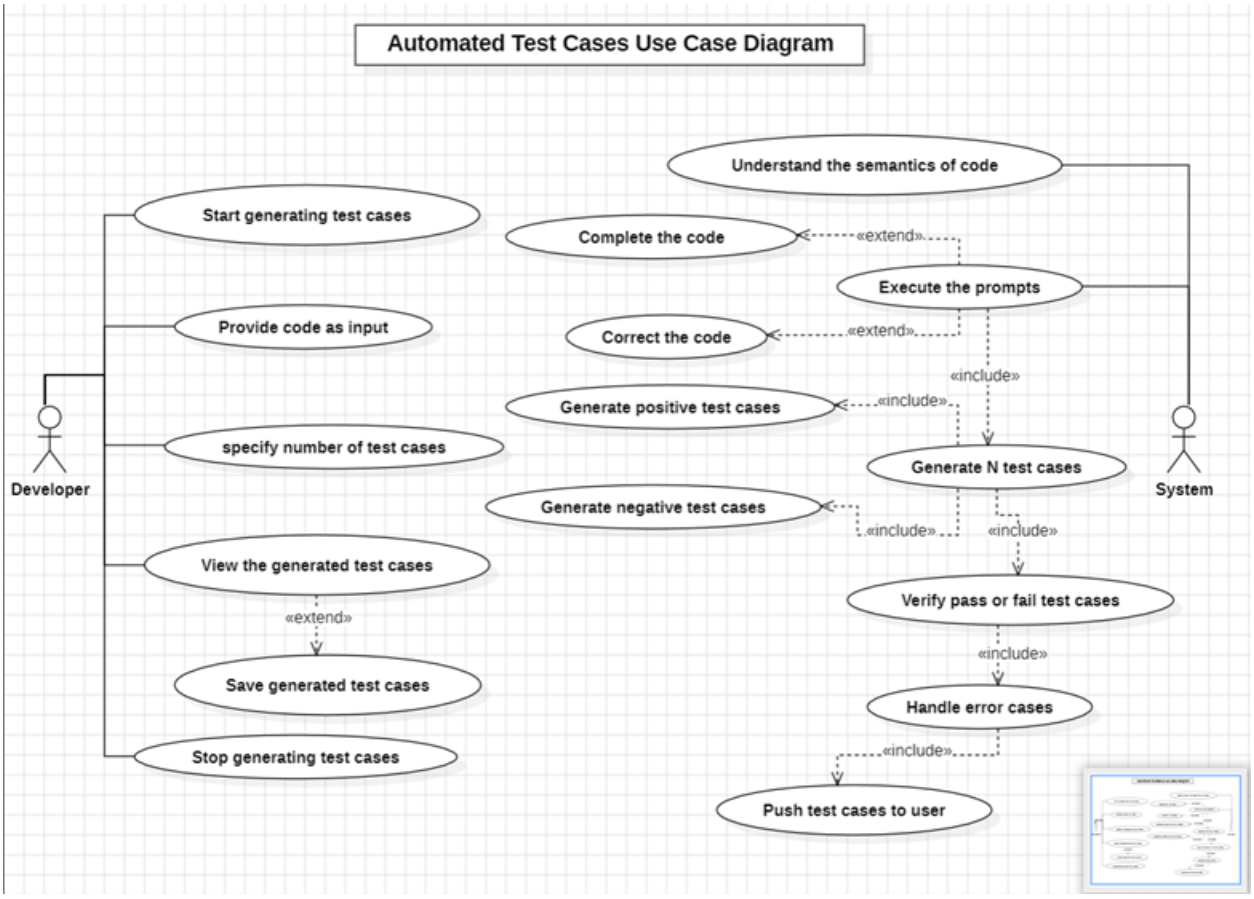

Figure 9: Test cases generation use case diagram.

2- Code Fault Localization:

Code fault localization is a critical process in software development that involves pinpointing the exact locations of defects within a codebase. This method automatically locates and reports the precise lines or code segments that include errors, giving comprehensive details like the line number and an explanation of the problem. After code modifications, the system makes sure that errors are quickly found and located. helping developers identify problems and find effective solutions. The system is also made to be secure, dependable, and scalable; it can accommodate large codebases and numerous users while upholding strict performance and data security guidelines. By making it possible to quickly identify and fix code flaws, this improves the overall quality of the program and eventually results in more reliable and strong applications. Show (Figure 10).

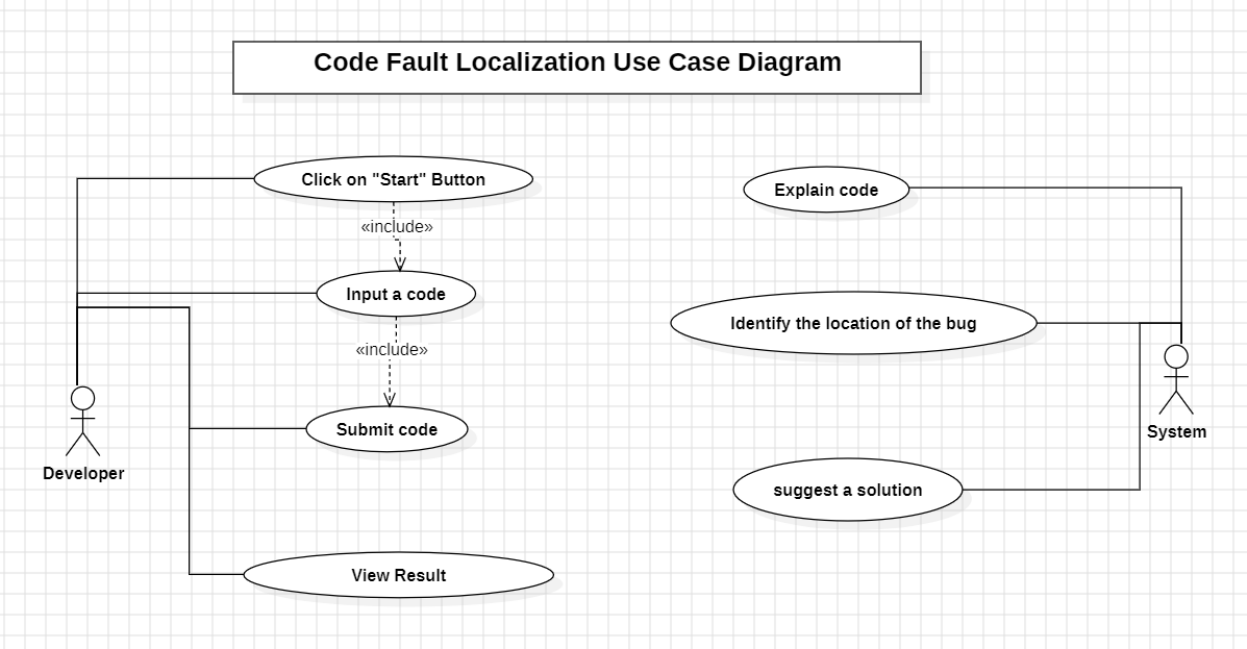

Figure 10: Code Fault Localization use case diagram

3.2.1 Functional Requirements:

3.2.1.1Automated Test Cases:

User Class 1 - The Developer

A. Functional Requirements:

Titel: Initiate Test Case Generation

Description: The system shall empower developers to commence the test case generation process by simply clicking the "Start Generating Test Cases" button.

B. Functional Requirements:

Titel: Input Code for Test Case Generation

Description: The system shall facilitate developers in entering code, enabling the generation of test cases specific to the provided code input.

C. Functional Requirements:

Titel: Specify the Number of Test Cases

Description: The system shall offer developers the flexibility to specify the desired number of test cases to be generated

D. Functional Requirements:

Titel: View Generated Test Cases

Description: The system shall provide developers with the capability to easily view and examine the test cases generated.

E. Functional Requirements:

Title: Stop Test Case Generation Process

Description: The system shall empower developers to interrupt the test case generation process at any given point by allowing them to stop the process.

System Class 2 – The System

A. Functional Requirements:

Title: Code Semantics Comprehension

Description: The system shall possess the capability to comprehend and understand the semantics embedded within the provided code

B. Functional Requirements:

Title: Prompt Execution for Test Case Generation

Description: The system shall adeptly execute prompts, employing them as directives for the systematic generation of test cases.

C. Functional Requirements:

Title: Code Autocompletion

Description: The system shall autonomously complete code segments when they are found to be incomplete, ensuring a comprehensive and valid code structure.

D. Functional Requirements:

Title: Code Autocorrection

Description: The system shall automatically correct code inaccuracies, providing a refined and error-free version of the code if errors or inconsistencies are detected.

E. Functional Requirements:

Title: Generate n Test Cases

Description: The system shall generate n test cases for each provided code, ensuring an adequate set of test scenarios for evaluation.

F. Functional Requirements:

Title: Generate Diverse Test Cases

Description: The system shall produce a diverse array of test cases, encompassing both positive and negative scenarios, thereby offering a comprehensive evaluation suite for the provided code.

G. Functional Requirements:

Title: Verify Test Case Outcomes

Description: The system shall systematically evaluate and verify the pass or fail status of each generated test case. This verification process is essential for understanding the behavioral outcomes of the test cases.

H. Functional Requirements:

Title: Error Case Handling

Description: The system shall adeptly handle error cases before initiating any communication with developers. This includes identifying and resolving errors within the generated test cases to ensure the delivery of accurate and reliable results.

I. Functional Requirements:

Title: Push Test Cases

Description: The system shall adeptly push the generated test cases to developer through interface.

3.2.1.2Code Fault Localization:

User Class 1 - The Developer

A. Functional Requirements:

Titel: Click on Start Button

Description: The system shall empower developers to commence the code fault process by simply clicking the "Start Code Fault Localization" button.

B. Functional Requirements:

Titel: Input Code

Description: The system shall facilitate developers in entering code, enabling debugging specific to the provided code input.

C. Functional Requirements:

Titel: Submit code

Description: The system shall provide developers with the capability to easily submit code for debugging process.

D. Functional Requirements:

Title: View Result

Description: The system shall empower developers to view the result of debugging process.

System Class 2 – The System

A. Functional Requirements:

Title: Explain code

Description: The system shall possess the capability to explain the code that developer entered.

B. Functional Requirements:

Title: Identify Location of the bug

Description: The system shall adeptly identify the location of the bugs by specifying the lines that contain bugs.

C. Functional Requirements:

Title: Suggest solutions

Description: The system shall autonomously generate solutions for the detected bugs to solve errors.

3.2.2 Software Interface:

The automated test cases website is designed to facilitate the generation of test cases for web applications. Utilizing the Llama-2 model, the system enables the creation of comprehensive test scenarios for web functionalities. The website operates within a web development environment. Automated test case generation is facilitated through an intuitive user interface, allowing developers to input code snippets for testing. The system interacts with the Llama-2 model to generate test cases based on the provided code. Test cases are designed to validate the functionality and behavior of web applications across various scenarios. Additionally, the website integrates with external APIs and services for enhanced functionality, such as fetching data from databases or interacting with backend systems. Error handling mechanisms are implemented to ensure smooth execution of test case generation tasks, providing developers with reliable testing solutions for web applications.

3.2.3 Network Interfaces:

The network interface within the automated test cases website serves as a crucial link between the web application and external resources, enabling seamless communication and data exchange. Utilizing standard HTTP and HTTPS protocols, the network interface facilitates interactions with backend servers and external APIs. Through the network interface, the website can efficiently send outgoing data, including requests for information or resources, such as code snippets or test case specifications, to backend systems.

3.2.4 System Interface:

We provide a web interface for consulting our application. When a developer tries to use our website, he/she would see two services in home page as in figure11 and figure 12. First Once clicking on the service you need, our system tries to generate test cases or de for each function after extracting it from the class and number of passed and detecting buggy code.

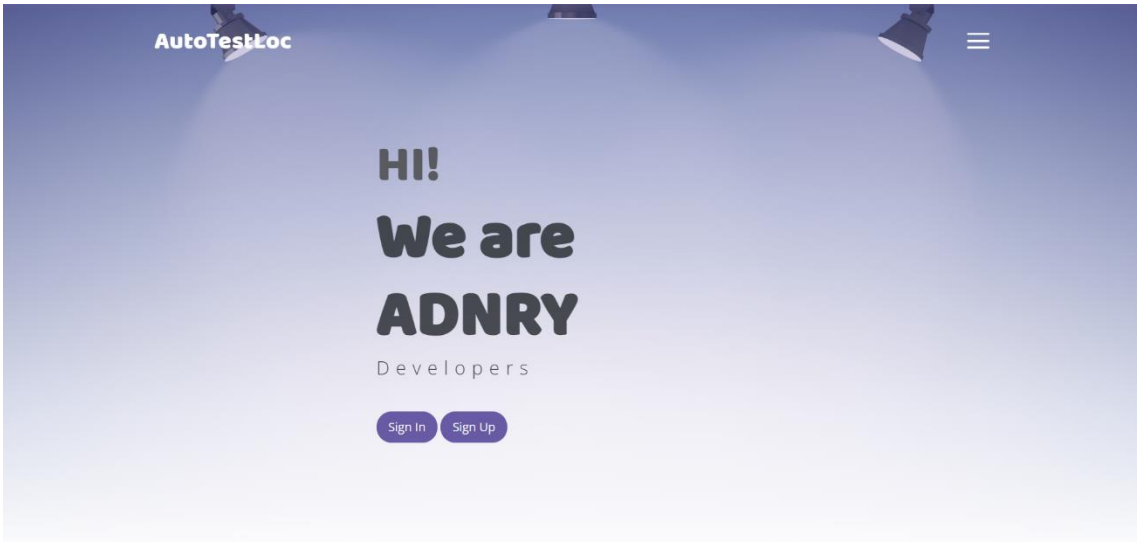

Figure 11:Home Page

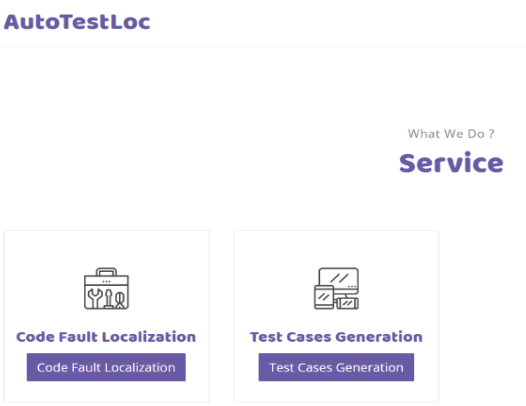

Figure 12: Website services

Web page for consulting our application. When a developer tries to use our website, he/she would see two boxes as in figure. First box for pasting or writing his/her own code or for dragging python file to detect it. Once clicking on submit button, our system tries to detect the buggy code with preparation for each function after extracting it from the correct code will appear also.it might be clearer to you that you should write your code input box and then correct code will appear in output box.

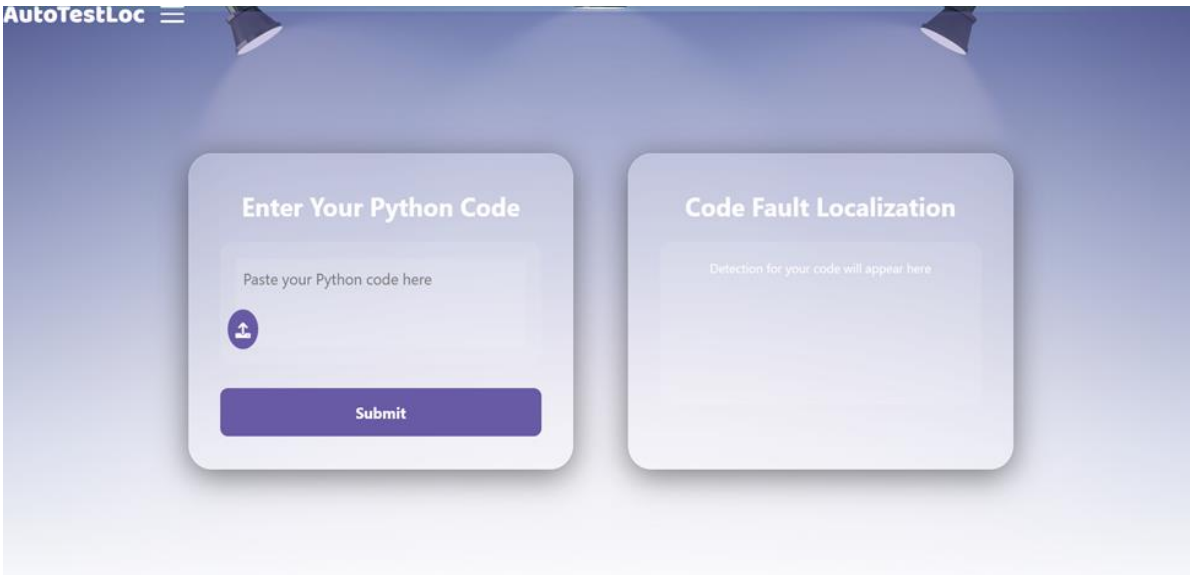

Figure 13: Code Fault Localization Page

Finally, when a developer tries to use our website, he/she would see two boxes as in figure. First box for pasting or writing his/her own code or for dragging python file for testing it. Once clicking on submit button, our system tries to generate test cases for each function after extracting it from the class and number of passed and failed cases will appear also.it might be clearer to you that you should write your code input box and then test cases will appear in output box

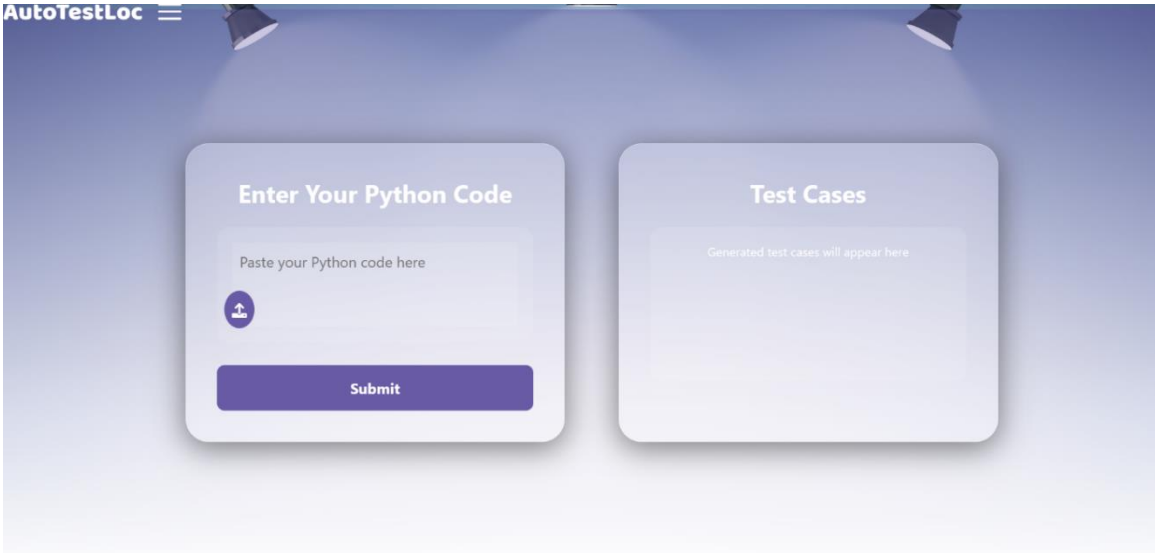

Figure 14: Test Case Page

3.2.5 Non-Functional Requirements:

Security: To protect sensitive data and guarantee the integrity and confidentiality of test data, source code, and execution results, the automated test cases project must abide by strict security guidelines. To stop unwanted access or tampering with important components, access controls and encryption techniques should be implemented.

Reliability: The maximum of three instances of Mean Time Between Failures (MTBF) annually. Robust error handling, version control, continuous monitoring, and automated testing of its own functionality are some of the ways the system accomplishes this. The reliability of the system is guaranteed by proactive steps such as comprehensive documentation and a dedication to quickly identifying and resolving problems. To facilitate smooth software development processes, the

automated test cases system reliably generates accurate and effective test cases with a focus on dependability.

Maintainability: Automated test cases should be designed with maintainability in mind, making it simple to add, update, and modify the test suite. Modular architecture, version control procedures, and well-documented code all contribute to the system's overall maintainability.

Portability: The system for generating test cases ought to be adaptable to various platforms, environments, and programming languages. This guarantees adaptability and simplicity in integrating with different development workflows and toolchains.

Extensibility: To enable users to quickly adopt new test case generation techniques, adjust to changing testing specifications, and easily integrate with emerging technologies, the system ought to facilitate extensibility. Adding unique test case generators or plugins should be made easier by the architecture.

3.3 Design Constraints:

- Software constraint that our website used language is python only limiting its applicability to other programming languages which means user should enters into the system python file.

- The amount of memory, processing power, and storage that users have access to may be restricted.

- The internet connection's dependability is essential. Inconsistent connections that frequently drop out or have slow speeds may affect the website's performance.

- The system is compatible with various operating systems (Windows, macOS, Linux) and browsers to ensure broad accessibility. This involves using cross-platform development frameworks and testing the system across different environments.

3.4 Research Design:

We aim to revolutionize software testing by developing an automated test case generation system that leverages the power of AI. Manual testing is time-consuming, prone to human error, and often provides limited coverage. Our goal is to enhance testing efficiency for developers by using large language models such as LLAMA and advanced machine learning methods to streamline the process. Our preferred model, LLAMA 2, is notable for being smaller, open-source, and flexible enough to be used in a wide range of applications. Its modular architecture allows for fine-tuning and customization, making it the optimal solution for our automated testing framework. Additionally, we are implementing code fault localization using LLAMA 3, which further improves our testing framework by identifying the exact location of defects within the code. This dual approach enables us to speed up the development cycle, increase defect detection, and improve software quality. We successfully attained our objectives by implementing the following key steps:

3.4.1 Automated test case generation:

3.4.1.1 Dataset:

We utilized the HumaneVal dataset, an extensive collection of 164 expertly designed problems. This dataset provides a consistent and standardized evaluation process by including comprehensive descriptions, inputs, expected results, and unit tests for each item. HumanEval covers a wide range of coding topics, offering insightful information on various coding skills and problems. These task categories include Code Comprehension, Algorithm Implementation, and Simple Mathematics. In addition, we used the code_contest_python3_alpaca dataset, which contains approximately 8.14k rows of Python code and their respective test cases. This dataset is available on Hugging Face and serves as a valuable resource for our automated testing framework, allowing us to further enhance the quality and coverage of our test case generation and fault localization efforts.

3.4.1.2 LLAMA-2 models:

Llama-2, a transformer-based LLM, includes an encoder for processing input code via self-attention layers and a decoder for creating output text, such as test cases. Layer normalization, residual connections, and positional encoding are all included. These techniques help Transformer models understand word order, train faster, and perform better. Due to its adaptable nature, it may be made more effective in a variety of tasks and domains by fine-tuning individual layers or components.

3.4.1.3 Fine tune:

To supervise the fine-tuning procedure, we hired a supervised fine-tuning trainer. The trainer plays a crucial role in helping the model learn task-specific patterns and characteristics from the given data. It increases convergence and improves performance by leveraging on existing knowledge. Using the training dataset as a guide, the trainer iteratively updates the model's parameters during the training process. When the training procedure is finished, the trained model is stored for later use, which saves the need to start the model over whenever user data needs to be predicted. We first fine-tuned the model on the HumaneVal dataset and saved the fine-tuned model. Afterward, we fine-tuned the model again on the code_contest_python3_alpaca dataset. This sequential fine-tuning process ensured that the model could benefit from both datasets, enhancing its ability to generate and evaluate test cases effectively.

3.4.1.4 Prompt & user input:

We created a personalized prompt that serves as a helpful tool to inform the model about how many test cases to generate depending on the problem. This helps the model in comprehending precisely what we need from it. Additionally, we included functionality that allows users to enter directly into the model. Users can upload code files for analysis, and the model will read and produce test cases. The model generates test cases with examples for both input and output. It also informs us of the success or failure of each test case.

3.4.2 Code Fault Localization:

3.4.2.1 Dataset:

We utilized the Python Bugs dataset, an extensive collection of 1,000 functions. This dataset includes columns for the corrected code, the prompt, and the task type. The Python Bugs dataset provides a rich resource for understanding and addressing various coding errors, offering valuable insights into common bug patterns and their solutions.

3.4.2.2 LLAMA-2 models:

LLAMA 3 leverages the transformer architecture, utilizing self-attention layers in its encoder to process input information. This design allows it to understand complex relationships within data, like how LLAMA 2 comprehended code structure. However, LLAMA 3 goes a step further with its advanced decoder, capable of generating not just text but potentially diverse creative formats, expanding its capabilities beyond mere test cases. Additionally, LLAMA 3 likely incorporates advancements such as improved layer normalization and enhanced residual connections, which contribute to faster training speeds and overall performance. This emphasis on adaptability indicates that LLAMA 3 can be fine-tuned even more effectively across various tasks and domains, making it a powerful tool for a wide range of applications.

3.4.2.3 Fine tune:

Fine-tuning allows the model to specialize in detecting and correcting code defects, which is significant in improving code fault localization. The model learns to accurately identify the context of bugs and acquires insights into common patterns by training on relevant datasets. This task-specific information improves the model's accuracy and efficiency in identifying errors, which raises the caliber of the software. To accelerate the fine-tuning process of our model, we utilized Unsloth, a specialized software suite designed to expedite training for large language models (LLMs). Leveraging advanced techniques, Unsloth achieves notable efficiency gains by reducing training time by up to 30 times and minimizing memory consumption by up to 60% compared to traditional methods. This streamlined training not only conserves resources but also empowers our

model to swiftly adapt and excel in identifying complex code issues, contributing to more robust and dependable software solutions.

3.4.2.4 Prompt & user input:

We used Alpaca Prompt to create a prompt that tells the model exactly what we want it to do, which is localize any kind of problem, logical or syntax related. The prompt directs the model to identify the exact line where the error occurs, provide a detailed explanation of the fault, and ultimately propose a fix for the faulty code. Like our automated test case generation system, this approach allows users to input code directly or upload it via a file. This structured approach ensures that the model not only identifies errors comprehensively but also offers actionable solutions, making the process of code fault localization more efficient and effective.

3.5 Architectural Design:

3.5.1 Architectural of the System:

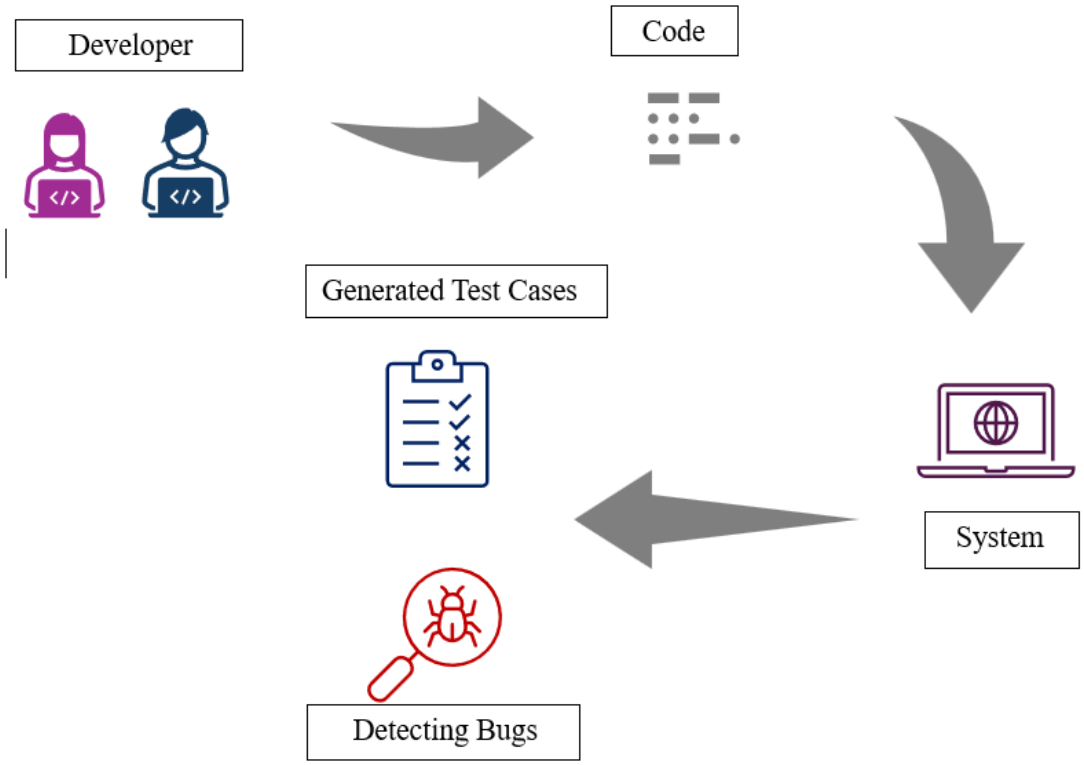

Figure 15: Architecture of the System

Figure 15 provides an overview of our system. To initiate any process, whether it's automated test case generation or code fault localization, the developer first needs to input their code into the system. Next, the developer selects the desired process: generating test cases or detecting bugs in the code.

If the developer opts to generate test cases, the system prompts them to specify the number of test cases required. Once specified, the system generates and presents these test cases to the developer. On the other hand, if the developer chooses to start the code fault localization process, the system analyzes the code, identifies bug locations by line, and suggests potential solutions.

Additionally, it provides an explanation of the code's functionality to help the developer understand the context of the identified issues.

3.5.2 Components Interaction:

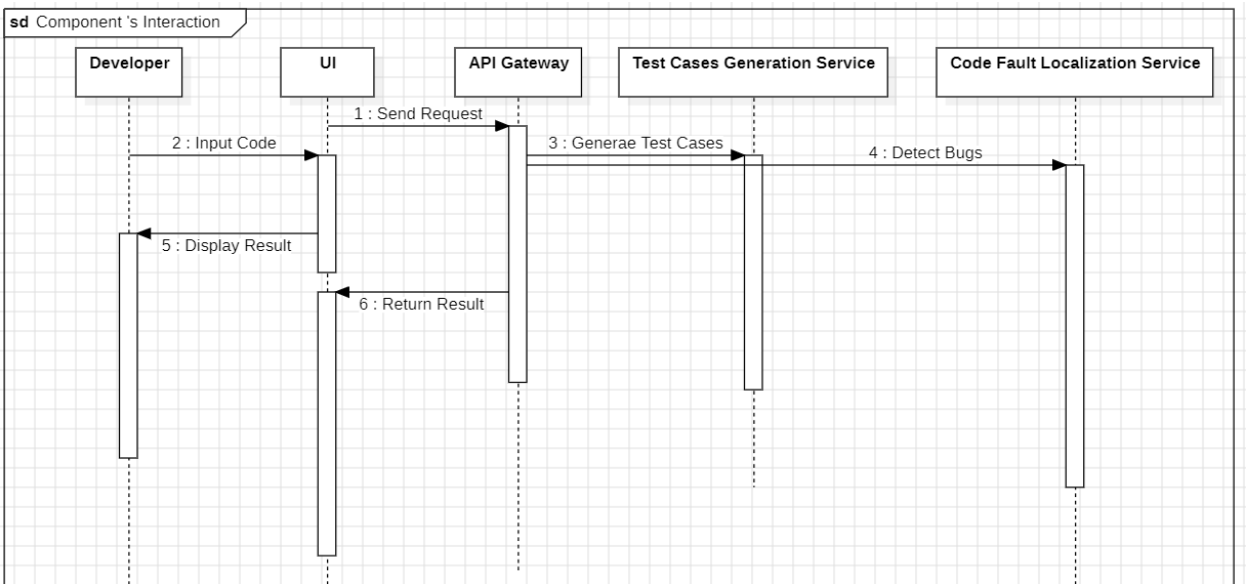

Figure 16: Sequence Diagram

The system consists of multiple essential parts that enable the automatic creation of test cases and the localization of code faults. Developers enter their code and choose the desired procedure on the User Interface (UI), which acts as the front-end platform. The API Gateway oversees handling UI requests and forwarding them to the relevant backend services. Based on the developer's instructions, the Automated Test Case Generation Service generates test cases; on the other hand, the Code Fault Localization Service locates and detects code problems. making sure of error control and performance. Together, these elements improve the efficiency and overall quality of the code by streamlining the development and debugging processes.

3.6 Component Design:

3.6.1 Large Language Models (LLMs)

The system employs two specialized Large Language Models (LLMs) to achieve its core functionalities: automated test case generation and code fault localization. These models leverage the advanced capabilities of deep learning to understand and process Python code effectively.

3.6.1.1 LLaMA-2 Model: Automated Test Case Generation

The LLaMA-2 model is specifically fine-tuned to generate automated test cases from Python code. This model is trained on a diverse dataset that includes various Python code patterns, functions, and their corresponding test cases. Here's a detailed breakdown of its functionality:

- **Understanding Code Patterns:** LLaMA-2 can recognize and understand different code patterns and structures, including loops, conditionals, functions, and classes. This enables it to generate relevant and comprehensive test cases that cover various execution paths.
- **Generating Edge Cases:** The model is capable of identifying potential edge cases based on the input code. This includes generating test cases for boundary conditions, exceptional inputs, and other scenarios that might not be immediately obvious.
- **Ensuring Code Coverage:** LLaMA-2 aims to maximize code coverage by generating test cases that cover all possible branches and paths within the code. This helps in identifying hidden bugs and ensuring the robustness of the code.

3.6.1.2 LLaMA-3 Model: Code Fault Localization

The LLaMA-3 model is designed for code fault localization, analyzing Python code to identify and pinpoint potential faults. This model is trained on a dataset containing examples of faulty code and their corrected versions. Here's a detailed breakdown of its functionality:

- **Syntax and Semantic Analysis:** LLaMA-3 performs both syntax and semantic analysis of the input code. It can detect syntax errors as well as logical errors that might not be

immediately apparent, such as incorrect calculations, misuse of variables, or improper control flow.

- **Contextual Understanding:** The model understands the context in which code is written, allowing it to provide more accurate fault localization. It can consider the surrounding code, dependencies, and the overall structure to identify faults effectively.
- **Detailed Feedback:** Upon identifying a fault, LLaMA-3 provides detailed feedback, including the exact location of the fault (line number and code snippet) and a description of the issue. This helps developers quickly understand and address the problem.
- **Suggestion for Corrections:** In addition to identifying faults, LLaMA-3 can also suggest potential corrections. These suggestions are based on common programming practices and patterns observed in the training data, providing developers with actionable insights to fix the code.

By leveraging these advanced LLMs, the system provides robust and reliable solutions for automated test case generation and code fault localization, significantly enhancing the efficiency and effectiveness of the development process.

3.7 Data Design:

Two key datasets are involved in programming test case development and code fault localization: the "python-bugs" dataset from OpenAI and the HumanEval dataset from OpenAI. Every dataset has a distinct function in improving LLM capabilities and guaranteeing the accuracy of code generation and error repair procedures.

The HumanEval dataset is a carefully selected collection of 164 programming tasks that are intended to make the assessment of code generation models easier. This dataset's tasks are organized with canonical answers, test cases, and task prompts as necessary components. Important fields such as {task_id}, `prompt{, `canonical_solution{, `test`, and `entry_point} offer detailed information required to comprehend and carry out the programming tasks. Through stringent sampling techniques, verification processes, and manual coding efforts, the dataset guarantees the tasks' integrity and, consequently, the selection of appropriate programming

challenges. This dataset contains securely collected and stored Python code, allowing for safe execution and effective assessment of code generation models.

The "python-bugs" dataset, which is designed for code fault localization and bug rectification, is a complementary addition to the HumanEval dataset. One thousand instances of Python code, along with the related bug patch, make up this dataset. Important fields like `prompt_code}, `correct_code}, `task}, and `index} help to specifically classify and uniquely identify each instance, which makes it easier for LLMs to be trained for bug detection and resolution. Through the dataset's focus on particular bug categories (such binary operator mistakes), LLMs are better equipped to quickly find and fix comparable problems in new code. The accuracy and dependability of code defect localization are greatly improved by this structured technique, which strengthens the robustness of code generation models in practical applications.

When combined, these datasets provide a strong basis for expanding the potential of big language models in software development applications. The "python-bugs" dataset improves these models' capacity to identify and fix problems, leading to an overall increase in code quality and dependability, while the HumanEval dataset guarantees that code generation models operate at peak efficiency by offering well-structured jobs. Together, these datasets allow academics and engineers to obtain more reliable software code defect localization and encourage creativity in automated test cases.

3.8 Algorithmic Design:

3.8.1 Automated Test Cases:

The llama project's algorithmic methodology for unit test case creation takes a methodical approach to covering different facets of the llama class. For every method, basic functionality test cases are first created by supplying common inputs and confirming anticipated results. Boundary value analysis is then used to examine edge cases and potentially troublesome situations. Test cases for exception handling are designed to make sure the llama class responds appropriately to unexpected inputs by raising and verifying exceptions when necessary. To isolate the unit of code under test, techniques involving external dependencies are also taken into consideration for the application of mocking and stubbing. To ensure that methods changing the state of the llama object

update it effectively, state-based testing is used. Overall, the technique places a strong emphasis on a thorough testing approach that includes a variety of test cases to guarantee the accuracy and resilience of the llama class methods.

1. Identify the target function or module to be tested.
2. Analyze the requirements and specifications of the target function or module.
3. Determine the possible input values and edge cases based on the requirements.
4. Generate test cases by combining different input values, including normal and boundary cases.
5. Write test case templates for each test scenario.
6. Use assertions or expected output values to define the expected behavior of the target function or module.
7. Execute the test cases and compare the actual output with the expected output.
8. Report the test results, including passed and failed cases.

In this pseudocode, we define a function generateTestCases that takes a targetFunction as input. It initializes an empty list called testCases. Then, for each possible input, it creates a test case, sets the input, and expected output, and appends the test case to the list. Finally, it returns the list of tests cases.

```
// Define a function to generate test cases for a given target function
function generateTestCases(targetFunction)
  // Initialize an empty list of test cases
  testCases = empty list
  // Loop through all possible inputs
  for each input in possibleInputs
    // Create a new test case object
    testCase = createTestCase()
    // Set the input and expected output of the test case
    setTestCaseInput(testCase, input)
    setTestCaseExpectedOutput(testCase, targetFunction(input))
    // Add the test case to the list
    appendTestCase(testCases, testCase)
  // Return the list of test cases
  return testCases
```

Figure 17 Pseud Code Generate Testcase

3.8.2 Code Fault Localization:

The primary algorithm used in this project involves leveraging Large Language Models (LLMs) for fault localization in code. The system utilizes advanced natural language processing techniques to understand the context and semantics of the code. It then compares the observed patterns with known fault patterns and provides localization of potential errors.

1- **Input Processing**:

-The source code is preprocessed to extract relevant features.

-Tokenization of the code into manageable pieces.

-Generation of an abstract syntax tree for structural analysis.

2- **Context Understanding**:

-The LLM analyzes the code to understand its context and semantics.

-Identification of variables, functions, control structures, and their interrelationships.

3- **Pattern Matching**:

-The system compares the analyzed code with a database of known fault patterns.

-Sophisticated pattern matching algorithms identify deviations and anomalies.

4- **Fault Localization**:

-Based on pattern matching results, the system localizes the faults.

-Detailed feedback on the nature and location of potential errors is generated.

5- **Output Generation**:

A comprehensive report is created, highlighting the faults and providing suggestions for correction

3.9 Interaction Design:

Our website allows users to choose between two main features: test case generation and code fault localization. Users interact with the program by uploading a file or directly entering a code. The website generates either a fixed code or test cases based on the user's choice. Our design emphasizes simplicity and user-friendliness, featuring an intuitive interface that is responsive and adaptive across various devices and screen sizes. This ensures seamless performance and usability, enhancing the overall user experience. Clear instructions and guidance are integrated throughout the interaction process, including inline text instructions, to help users provide accurate inputs and configure parameters effectively. By prioritizing ease of use and providing robust functionality, our website caters to both novice users and experienced developers, facilitating efficient software testing and code refinement processes with clarity and precision.

3.10 Experimental Setup:

Using the Llama-2 and Llama-3 models for automated test case generation and code fault localization is a methodical software testing technique that makes use of cutting-edge natural language processing capabilities. In the Colab GPU environment, the experimental setup consists of setting up the Llama-2 and Llama-3 models for fine-tuning on a collection of test cases and examples of Python functions. To maximize model performance and training, hyperparameters including batch size, learning rate, and optimization approach are carefully adjusted. To ensure that the dataset is compatible with the model, tokenization and normalization procedures are included in the preprocessing phase. Evaluation metrics are developed to evaluate the relevance and quality of generated test cases. These metrics include code coverage, assertion completeness, and functional correctness. Strict validation procedures are used throughout the experiment to guarantee the robustness and dependability of the findings. The experimentation approach aims to demonstrate the effectiveness and usefulness of the automated test case generation and code fault localization system driven by the Llama-2 and Llama-3 model by utilizing the computational capabilities offered by Google Colab GPU.

Chapter 4:

Implementation and Preliminary Results

4.1 Programming Languages and Tools:

Python is a dynamically typed, high-level programming language that follows an interpreted paradigm. It emphasizes code readability and conciseness, making it an accessible language for both beginners and experienced developers. With its rich standard library and community support, Python is widely used in various domains, including web development, data science, artificial intelligence, and more. The interpreter-based execution allows for quick and flexible development, fostering a balance between simplicity and powerful functionality. Web development, scientific computing, data analysis, and artificial intelligence are just a few of the many uses for it. Because it includes many tools and frameworks that make it simple to create and train LLMs, Python is a popular choice for deep learning. These libraries TensorFlow, HuggingFace, OpenAI and PyTorch, for example, offer a high-level interface for creating and refining deep learning models, enabling the construction of intricate large language models with a small amount of code. Furthermore, a sizable and vibrant Python user and development community implies a wealth of resources for learning and debugging. Deep learning applications can be effortlessly integrated with the language's abundance of pre-processed datasets and pre-trained models.

4.1.1 Jupyter Notebook:

Jupyter Notebook is an open-source interactive online tool that lets you create and share documents with narrative text, equations, live code, and visualizations. It is a multipurpose tool for data analysis, machine learning, and scientific research since it supports several computer languages, including Python, R, and Julia.

4.1.2 TensorFlow:

The Google Brain team created the open-source machine learning library TensorFlow. With a focus on deep learning, it offers a complete platform for developing and implementing machine learning models. An environment as adaptable and versatile as TensorFlow can handle a variety of jobs, from straightforward linear regression to intricate neural network topologies.

4.1.3 PyTorch:

Popular for its adaptability and dynamic computational graph, PyTorch is an open-source machine learning framework for Python that can be used for both production and research. In addition to capabilities for automated distinction, optimization, and effective GPU support, it offers a framework for creating and training neural networks. Many fields, including computer vision, reinforcement learning, and natural language processing, use PyTorch extensively.

4.1.4 Hugging Face:

A Python package called Hugging Face offers a wide range of natural language processing (NLP) capabilities and trained models. The Transformers library, which provides quick access to a variety of cutting-edge pre-trained language models, including those built on OpenAI's GPT architecture, is one of its primary features. These models may be utilized straight out of the box for a variety of applications including text creation, and generation, or they can be adjusted for NLP tasks.

4.1.5 OpenAI:

Especially for big language models like Llama 2, the OpenAI Python module simplifies communication with the API and provides a smooth integration for automated unit test case development. It guarantees type safety and minimizes mistakes with well-defined parameter definitions and answer forms. Its synchronous and asynchronous features maximize testing effectiveness and improve project security through secure key management via environment variables. Use the OpenAI Python library to improve your unit testing and make use of Llama 2's strong LLM features to test your projects more thoroughly and quickly.

4.2 Code Structure:

To advance our project, we've successfully navigated through four pivotal steps:

4.2.1 Input preprocessing:

The user input is facilitated through two methods: manual entry or file upload. Users can input data either by typing it directly or by uploading a file. This dual approach allows for flexibility and convenience. Our system seamlessly handles multiple Python functions entered by the user. Whether the input comprises a single function or multiple functions, our system divides and processes them individually. This ensures thorough analysis and optimization of each function.

4.2.2 Test case generation:

Initially, we loaded the base model and tokenizer. Then we applied a customized setup called Quantized LoRA. This setup makes it possible to include quantization into the model, which lowers the model's weight and activation precision, often to 4-bit precision. LoRA is a form of attention mechanism that determines whether parts of the input sequence are relevant enough to produce tokens as output. Memory and computational efficiency are improved as a result of this optimization. We trained the model on the HumanEval and code contest python3 alpaca datasets and then the model was saved for future use. To generate test cases effectively, we initialize the process by inputting a constant prompt, specifying the desired number of test cases, and defining the exact shape of the output. Subsequently, we indicate the task for the model and constrain the maximum length to 1000 tokens. This systematic approach ensures precise control over the test case generation process, facilitating accurate and efficient model performance evaluation.

4.2.3 Code Fault Localization:

Similar to our approach for test case generation, we utilized LLAMA to load the base model. We utilized unsloth to load the model and the tokenizer. Unsloth supports 4-bit quantization, which enables the model and tokenizer to load faster and occupy less space. Unsloth employs various techniques to enhance overall training efficiency, including Flash Attention which is a method that helps the LLM focus on the most critical parts of the input data, and quantization. After training on the Python Bugs dataset, the model was uploaded to Hugging Face. Then, for sufficient memory use, it is reloaded using 4-bit quantization. To generate code fault localizations, we use a prompt that defines the structure of the output. This prompt includes an instruction detailing what the model should do, along with the input and the expected output.

4.2.4 Output processing and validation:

Upon generating test cases, the model's output is promptly presented to the user. Additionally, we've engineered a dedicated function designed to extract these test cases from the model's output. This function plays a pivotal role in assessing the success or failure of each test case. By systematically comparing the model-generated tests against predefined criteria, we ascertain

whether the tests have passed or failed, ensuring the reliability and effectiveness of the generated test suite. For code fault localization, the model's output is similarly detailed and user-friendly. The model identifies the error line and type, provides an explanation for the fault, and then rewrites the corrected code. This comprehensive approach ensures that users not only understand where and what the error is but also why it occurred and how to fix it. By presenting both the identified issues and their solutions, our system greatly enhances the user's ability to refine and debug their code efficiently.

4.3 Data Structures and Databases:

In this project, the primary focus is on utilizing large language models (LLMs) to automate test case generation and code fault localization. Specifically, two models are employed: LLaMA-2 for generating automated test cases and LLaMA-3 for pinpointing faulty code within Python scripts. Given the reliance on these advanced models, the use of traditional data structures is minimal. Instead, the models handle most of the computational and analytical tasks directly. For data management, a relational database is used to store user data, code submissions, test case results, and fault localization outputs. This database is designed to ensure efficient data retrieval and integrity, supporting the system's operational needs. Additionally, the system may use a NoSQL database to manage logs and other unstructured data, providing the necessary flexibility and scalability. This database infrastructure ensures that all data interactions are seamless and efficient, supporting the high-level operations performed by the LLMs. The schema includes tables such as "TestCases" to store details of generated test cases, "CodeSnippets" to store the input code provided by users, and "FaultLocalizationResults" to record the outputs generated by the LLaMA-3 model indicating the identified faults. Additionally, tables like "Users" handle user authentication and management, while "Configurations" store system settings. Relational tables, such as "TestCaseResults" to track the outcomes of executed test cases and "CodeSubmissions" to link users with their respective code submissions and results, ensure robust data relationships. This structured approach facilitates efficient data retrieval, integrity, and scalability, supporting the system’s core functionalities of automated test case generation and precise code fault localization.

4.4 Quantitative Results:

The table displays the findings of a test on producing unit test cases for a program using the Llama language model. Twenty sample inputs were utilized in the experiment, and the following table displays the findings:

We tried entering a different number of functions with each iteration.

Iteration	Input	Pass	Fail
1	1	3	0
2	5	15	0
3	10	30	0
4	15	44	1
5	20	58	2

We concluded that by increasing number of functions accuracy decreases.

As you can see, all 20 sample inputs have unit test cases that accurately predicted the programme's result from Llama language model. This implies that the model might be a helpful resource for creating unit test cases that test pass 58 test case and only 2 test cases was failed. Overall, the table demonstrates that precise and useful unit test cases might be produced by the Llama language model. Although this is a promising result, additional research is required to validate these conclusions.

In the evaluation of our code fault localization approach, we used “code_eval” technique to measure accuracy of model’s output code if it correctly recommend correct code or not so, used QuixBugs dataset that has 50 python buggy files and has its test cases. So, we test each output

generated with it test cases. If any output tested failed so the output is incorrect. We observed that the model got 34 correct and 16 wrong. You could observe that in our table below.

Python file name	**Fix Successfully**
Bitcount	✓
breadth-first-search	✓
Bucketsort	✓
depth-first-search	✗
detect-cycle	✓
find-first-in-sorted	✓
find-in-sorted	✓
Flatten	✓
Gcd	✓
get-factors	✗
Hanoi	✓
is-valid-parenthesization	✓
Kheapsort	✗
Knapsack	✓
Kth	✗
lcs-length	✓
Levenshtein	✓
Lis	✗
longest-common-subsequence	✗
max-sublist-sum	✓
Mergesort	✓
minimum-spanning-tree	✓
next-palindrome	✗

next-permutation	✓
Pascal	✓
possible-change	✓
powerset	✓
quicksort	✗
reverse-linked-list	✓
rpn-eval	✓
shortest-path-length	✓
shortest-pathlengths	✗
shortest-paths	✗
shunting-yard	✓
sieve	✓
sqrt	✓
subsequences	✓
to-base	✓
topologicalordering	✗
wrap	✓

We concluded that once python file has a lot of classes and function the test cases fails. It demonstrated effectiveness in recommending correct code for a significant portion of the cases. While there were instances where the model's recommendations were incorrect, the overall performance indicates potential for use in code fault localization model. Further refinement and analysis of failure cases could enhance the model's accuracy and broaden its applicability in real-world scenarios.

4.5 Qualitative Results:

Following the integration of the Llama model into our automated test case generation system, we've observed an impressive overall accuracy rate of 96% determined through human evaluation. Notably, the model exhibits optimal performance when handling a smaller number of functions. Furthermore, our observations revealed that the model excels in covering diverse test case scenarios, including the evaluation of null inputs. The implementation has also resulted in substantial time savings, empowering our team to delve deeper into test result analysis with heightened focus and efficiency.

Integrating our code fault localization model into our development pipeline has been a game-changer. While the model's 68% accuracy leaves room for improvement, its impact on our workflow has been significant. It's been instrumental in quickly identifying and resolving a majority of code faults, leading to enhanced software quality. Moreover, the model has provided valuable insights into complex bugs and code patterns, informing our efforts to refine our development practices and code review processes for even better outcomes. Overall, LLaMA-3 has proven to be a powerful tool in our arsenal, significantly improving our software development process and the quality of our code.

Chapter 5:

Discussion

5.1 Interpretation of Results:

Through leveraging the advanced capabilities of LLM, we have successfully achieved our objectives of automating both test case generation and code fault localization. Our system is adept at accepting individual functions or multiple functions as input, tailoring the generation of multiple test cases to each function's specifications, or pinpointing errors within the code and proposing effective fixes. Users benefit from the flexibility of our system as it can accept one or more functions. The system then provides insights by evaluating the output. For test case generation, our system produces assert statements that enumerate the number of test cases generated for each function. This clear output aids in verifying the correctness of the implemented functions across diverse scenarios. In the context of code fault localization, our system identifies the specific line of error, categorizes the type of error encountered, and provides clear explanations of the nature of the error. Additionally, it offers actionable fixes to rectify the identified faults. This comprehensive approach ensures that users not only understand the errors within their code but also have the tools necessary to effectively resolve them, thereby enhancing overall code quality and reliability. Our system automates comprehensive test case generation tailored to each function, alongside precise code fault localization that identifies errors, categorizes types, and provides actionable fixes. It handles flexible inputs for both individual and multiple functions, evaluating outputs to enhance code reliability and quality effectively.

5.2 Comparison with Previous Studies:

Recent research using large language models (LLMs) has significantly boosted software testing and fault localization. As detailed in LLM studies, these new approaches promise greater accuracy, consistency, and applicability, fundamentally reshaping these critical areas of software development.

5.2.1 Automated Test Cases:

5.2.1.1 Findings

Our research:

-The system leverages the Llama-2 model to generate comprehensive test cases for web applications. It features an intuitive interface for developers to input code snippets and receive generated test cases. The system also integrates with external APIs for enhanced functionality.

-The system aims to increase the efficacy and efficiency of software testing by automating the generation of unit test cases for software modules using predefined criteria and code analysis.

The previous study:

Discusses an approach that generates test cases automatically from bug reports, leveraging natural language processing (NLP) techniques to extract relevant information and create test cases. This system is designed to improve the detection and fixing of bugs by automating the creation of regression test cases from bug descriptions

5.2.1.2 Highlight Similarities

Both systems converge on the idea of automating test case generation, a strategy to expedite and strengthen the software testing process. To achieve this, they wield sophisticated models and techniques: Our research leverages Llama-2, while its counterpart harnesses the power of Natural Language Processing (NLP). This focus on automation resonates with the growing emphasis on integrating such functionalities seamlessly within the software development lifecycle. Achieving comprehensive testing coverage while minimizing the need for manual test case creation, a significant time saver for development teams.

5.2.1.3 Differences

Our research:

More focused on leveraging LLMs for automated test case generation, while the empirical paper discusses a broader range of empirical methods and their impact on test effectiveness and generate cases.

The previous study:

The empirical paper provides a broader theoretical framework, including various machine learning models and empirical data, compared to the specific focus on LLMs in Automated Test Cases.

5.2.1.4 Advancements

Our research:

Advances in using LLMs like Llama-2 for generating test cases from code, focusing on web functionalities and seamless integration with various APIs and services

The previous study:

Advances in using NLP for interpreting bug reports to generate targeted test cases, ensuring the relevancy of tests to the reported issues

5.2.2 Automated Test Cases:

5.2.2.1 Findings

Our research:

Discusses code fault localization as a process that involves pinpointing the exact locations of defects within a codebase. It highlights the use of the Llama-3 model for this purpose.The system aims to quickly identify and locate errors, providing detailed information like line numbers and problem descriptions .

The previous study:

focuses on the application of large language models (LLMs) for fault localization in code. It emphasizes the use of LLMs to analyze code and identify potential faults effectively.The methodology involves training models on a large dataset of code snippets and bug reports to enhance their fault detection capabilities

5.2.2.2 Highlight Similarities

Enhanced Accuracy and Efficiency: Both papers demonstrate how large language models (LLMs) can significantly improve the accuracy and efficiency of code fault localization.

Detailed Fault Reporting: A key focus for both approaches is providing developers with comprehensive fault information, including specific line numbers and explanations. This empowers developers to pinpoint and resolve issues swiftly.

Seamless Integration: An important advantage of both methods is their seamless integration into existing development workflows. This allows developers to leverage fault detection and correction capabilities without disrupting their established processes.

5.2.2.3 Differences

Our research:

specifically mentions the use of the Llama-3 model, whereas the previous research discusses LLMs in a more general context without specifying a particular model.

The previous study:

The former includes user class requirements and functional requirements specific to the system interface for code fault localization, while the latter focuses more on the overall methodology and training of LLMs.

5.2.2.4 Advancements

Our research:

Advances in integrating fault localization into a broader automated testing framework, emphasizing security, reliability, and scalability.

The previous study:

Advances in utilizing continuous model training and user feedback to improve the accuracy and efficiency of fault localization using LLMs.

5.3 Limitations:

There are various constraints that affect the AutoTestLoc tool's relevance and performance. Primarily, its emphasis on Python limits its application to applications created using this language. Reaching 100% test coverage can be difficult because there are likely to be untested portions of the code, especially in edge situations and infrequently used paths. Additionally, it can be challenging to automatically test some types of code, such as hardware interfaces or GUI elements. Fault localization can become less efficient and more difficult to implement in large and complicated codebases.

Chapter 6:

Conclusion and Future Work

6.1 Summary of Findings:

By leveraging advanced LLM capabilities, our system has effectively automated both test case generation and code fault localization. The primary findings are:

- **Automated Test Coverage:** the system successfully generates multiple test cases for each function. These test cases include assert statements that validate the correctness of functions across various scenarios.
- **Code fault localization:** The system identifies the exact lines where errors occur and categorizes the type of errors. It also provides detailed explanations and actionable fixes, facilitating efficient error resolution.
- **Flexibility and Versatility:** The system is capable of handling both individual and multiple functions as inputs, and it offers versatile usage. Each function is processed independently, ensuring precise and relevant test cases and error diagnostics.
- **Insightful Output Evaluation:** The system evaluates outputs to determine the number of passed and failed test cases.
- **Improved Code Quality:** By equipping users with tools to understand and resolve errors, the system significantly enhances overall code quality and reliability.

6.2 Future Work:

We are organizing a thorough expansion campaign to increase the functionality and reach of our Automated Test Case Generation and Code Fault Localization Tool. The objective of this next effort is to support a wide range of software projects through the implementation of many strategic improvements:

- **Multilingual Support:**

We want to incorporate support for many programming languages into the tool to make it flexible enough to be used in a variety of software projects. This will guarantee that the tool works with various codebases and development environments.

- **Use of Diverse Dataset:**

We want to fine-tune our model utilizing a wider and more diverse dataset in order to increase the tool's resilience in managing real-world coding styles and issues. This data will be sourced from platforms like Codeforces, which provide a variety of coding situations and difficulties.

- **Improve Module for Localizing Code Fault:**

Our plan includes developing a more sophisticated code fault localization module with the following essential features:

A. **Multi-Function Analysis:** By making improvements, the module will be able to operate on several functions at once, taking care of their interdependencies and defects as a whole.
B. **Multilingual Understanding**: To successfully handle and comprehend a variety of programming languages, the module will be enhanced.
C. **Optimization for right Code:** The module will further optimize code to improve accuracy and lower resource consumption, such as memory and time, when it receives the right code.
D. **difficult Error Resolution:** The tool's ability to recognize and correct increasingly complicated problems will increase its usefulness in difficult and sophisticated coding situations.

- **Accuracy and Assessment:**

We're dedicated to making sure the test cases we generate adhere to the strictest guidelines. Our development method will be focused on implementing trustworthy assessment techniques and improving accuracy on a constant basis. This emphasis will guarantee that the tool produces accurate and reliable test cases.

Our future work will focus on these important areas to create a more robust and inclusive tool that can adapt to changing requirements for software testing in a variety of scenarios and programming languages.

References

[1] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, & Qing Wang. (2024, January 5). *Software Testing with Large Language Models: Survey, Landscape, and Vision*. arXiv:2307.07221v2. https://arxiv.org/pdf/2307.07221.pdf

[2] Dakhel, A. M., Nikanjam, A., Majdinasaba, V., Khomh, F., & Desmarais, M. (2023, August 31). *Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing*. Arxiv. https://arxiv.org/pdf/2308.16557.pdf

[3] RYAN, G., JAIN, S., SHANG, M., WANG, S., MA, X., KRISHNA, M., RAMANATHAN, & RAY, B. (2024, January 31). Code-Aware Prompting: A study of Coverage guided Test Generation in Regression Setting using LLM. https://arxiv.org/pdf/2402.00097.pdf

[4] Bartram, D. (2015, March 7). *Test automation - past, present and future*. SlideShare. https://www.slideshare.net/Bugler/test-automation-past-present-and-future

[5] Yang, A. Z. H., Goues, C. L., Martins, R., & Hellendoorn, V. J. (2023, October 3). Large Language Models for Test-Free Fault Localization. https://arxiv.org/pdf/2310.01726.pdf

[6] Kang, S., An, G., & Yoo, S. (2023, August 26). A Preliminary Evaluation of LLM-Based Fault Localization. https://arxiv.org/pdf/2308.05487.pdf

[7] Brown, T. B., Mann, B., Ryder, N., & Subbiah, M. (2022, July 22). Language Models are Few-Shot Learners. https://arxiv.org/pdf/2005.14165.pdf &hellip

[8] Rogers, V., Meara, P., Barnett-Legh, T., Curry, C., & Davie, E. (2017, August 1). *Examining the llama aptitude tests.* Journal of the European Second Language Association. https://euroslajournal.org/articles/10.22599/jesla.24

[9] Plein, L., Ouédraogo, W. C., Klein, J., & Bissyandé, T. F. (2023, October 10). Automatic Generation of Test Cases based on Bug Reports: a Feasibility Study with Large Language Models. https://arxiv.org/pdf/2310.06320.pdf

[10] Touvron, H., Martin, L., & Stone, K. (2023, July 19). Llama 2: Open Foundation and Fine-Tuned Chat Models. https://arxiv.org/pdf/2307.09288.pdf

[11] Bartram, D. (2015, March 7). *Test automation - past, present and future*. SlideShare. https://www.slideshare.net/Bugler/test-automation-past-present-and-future

[12] Shamshiri, S. (n.d.). *Automated Unit Test Generation for Evolving Software*. UniBg. https://cs.unibg.it/esecfse_proceedings/fse15/p1038-shamshiri.pdf

[13] Zeng, M., Wu, Y., Ye, Z., & Xiong, Y. (n.d.). fault localization via efficient probabilistic modeling of program semantics . https://www.semanticscholar.org/paper/Fault-Localization-via-Efficient-Probabilistic-of-Zeng-Wu/8592c6bd234bed2517fbdad9221a766c015ad0fe

[14] Yang, A. Z. H., Goues, C. L., Martins, R., & Hellendoorn, V. J. (n.d.). Large Language Models for Test-Free Fault Localization. https://arxiv.org/pdf/2310.01726

[15] HOSSAIN, S. B., JIANG, N., ZHOU, Q., LI, X., CHIANG, W.-H., LYU, Y., NGUYEN, H., & TRIPP, O. (n.d.). A Deep Dive into Large Language Models for Automated Bug Localization and Repair. https://arxiv.org/pdf/2404.11595

[16] Li, Y., Wang, S., & Nguyen, T. N. (2021, February 27). Fault Localization with Code Coverage Representation Learning. https://arxiv.org/pdf/2103.00270v1

[17] KANG, S., AN, G., & YOO, S. (n.d.). A Quantitative and Qualitative Evaluation of LLM-Based Explainable Fault Localization. https://coinse.github.io/publications/pdfs/Kang2024ay.pdf