AN EXPERIMENTAL STUDY OF AUTOMATED TEST CASE GENERATION WITH LLMS

PUSPANJALI SARMA

Final Thesis Report

JUNE 2024

DEDICATION

This research proposal interim report is dedicated to all those who have inspired and supported me on this journey. To my family, whose unwavering love and encouragement have been my anchor through the highs and lows of this endeavour. To my friends and mentors, whose guidance and wisdom have enriched my understanding and fuelled my passion for research. And to all the individuals who have shared their knowledge and expertise, shaping the path of this project. Your belief in me has been a driving force, and I dedicate this work to each of you with heartfelt appreciation.

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to all those who have contributed to the development and progress of this research proposal interim report. Special thanks to my supervisor Dr. Kuncham Sreenivasa Rao, Associate Professor, Department of Computer Science and Engineering, ICFAI Foundation for Higher Education (IFHE), Hyderabad for their guidance, support, and invaluable feedback throughout this process. I am also grateful to Liverpool John Moore University, Liverpool, UK and Upgrad Education, India for providing resources and facilities essential for conducting this research. Additionally, I extend my appreciation to my colleagues and peers for their insightful discussions and encouragement. Their input has been instrumental in shaping the direction of this research. Lastly, I would like to acknowledge the unwavering support of my family and friends, whose encouragement has been a constant source of motivation.

ABSTRACT

Recently, pre-trained large language models (LLMs) have emerged as a ground-breaking technology in the fields of natural language processing and artificial intelligence. These models demonstrate exceptional performance across a wide range of tasks and possess the capability to handle large-scale datasets effectively. Concurrently, software testing stands as a crucial undertaking, playing an essential role in ensuring the quality and reliability of software products. As the scope and complexity of software systems continue to expand, the demand for more successful software testing techniques becomes increasingly urgent. This creates an opportune area for advanced approaches, such as incorporating LLMs.

This research work delves into the utilisation of Large Language Models (LLMs) within the realm of automated test case generation. Covering various aspects such as background information, problem statement, research inquiries, aims and objectives, significance of the study, scope, research methodology, and necessary resources, the work offers a thorough examination. Furthermore, it provides a detailed research plan to steer the implementation of the study.

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

LLM……...…. Large Language Model

LM……...Language Model

ATGC….....…...Automated Test Case Generation

GPT……..... Generative pre-trained

QA……...Quality Assurance

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 Background

Automated Test Case Generation with Large Language Models (ATCG) involves utilizing these sophisticated AI models to automatically create test cases for software applications. LLMs, equipped with a deep understanding of natural language and programming concepts, can analyze software requirements (E. Arteca, 2022), specifications, and code snippets to generate test cases that validate the functionality, robustness, and edge cases of the software. By comprehending the intended behaviour of the system and its potential usage scenarios, LLMs can generate test cases that cover a wide range of conditions, reducing manual effort and enhancing test coverage.

However, Automated Test Case Generation with LLMs encounters several challenges(Mirshokraie et al., 2015). Firstly, accurately understanding complex software requirements and specifications from natural language descriptions can be challenging due to ambiguities (Zalewski, 2023), implicit constraints, and varying interpretations. Moreover, generating test cases that cover a comprehensive range of scenarios(Saxena et al., 2010) while avoiding redundancies requires a deep understanding of the software's functionality and potential edge cases. Additionally, ensuring that the generated test cases are efficient, relevant, and comprehensive demands careful consideration of the system's behaviour under various conditions.

To surmount the hurdles in Automated Test Case Generation with LLMs(Yu et al., 2022), various strategies can be adopted. These include fine-tuning LLMs with domain-specific data pertinent to software testing, alongside integrating specialized architectures that focus on comprehending software requirements(McMinn, 2011; Panichella et al., 2018) and code structures. Such measures can augment the model's capacity to produce precise and diverse test cases. Implementing feedback mechanisms involving human validation and continual refinement can elevate the quality of generated test cases through iterative learning from the outputs(Fard et al., 2015). Additionally, merging LLMs with automated techniques (al., 2020) for test case prioritization and optimization can ensure the creation of impactful and efficient test suites, tackling the challenges of comprehensiveness and efficiency in test case generation. In summary, a blend of domain-specific training(Artzi et al., 2011), model enhancements, iterative refinement processes, and judicious utilization can mitigate the obstacles associated with Automated Test Case Generation using LLMs. (Siddiqui, 2021) We will explore the

potential of large language models such as GPT or StarCoder in automatically generating test cases for software applications, with a specific emphasis on evaluating the effectiveness, coverage, and efficiency of the generated test cases(Mastropaolo et al., 2023; Tufano et al., 2022).

1.2 Problem Statement

The ever-growing complexity of software applications, coupled with the increasing demand for rapid and reliable testing, poses a significant challenge in the field of software engineering (J. Shore and S. Warden, 2021) (K, 2000) (Siddiqui, 2021). Traditional methods of manual test case generation are not only labour-intensive but also struggle to keep pace with the dynamic nature of modern software (E. Daka and G. Fraser, 2014). Despite advancements in automated testing, existing approaches often fall short of achieving comprehensive effectiveness, coverage, and efficiency in the generation of test cases.

Current automated test case generation techniques, while providing some level of assistance, frequently lack the adaptability and contextual understanding required to thoroughly exercise the intricate functionalities within diverse software applications(Panichella et al., 2020). This limitation results in suboptimal test coverage, leaving potential vulnerabilities and undiscovered bugs (M. M. Almasi, n.d.) (Zalewski, 2023) (G. Grano, 2018) (A. Panichella, 2020) (E. Daka, 2015). As software systems continue to evolve in complexity, a critical need arises for innovative approaches that can enhance the efficacy of test case generation, ensuring a thorough examination of software functionalities.

In this context, the utilisation of Large Language Models (LLMs) like GPT (Generative Pre-trained Transformer) and StarCoder presents a promising avenue for addressing the shortcomings in automated test case generation (al., 2022) (S. K. Lahiri et al., 2022). These sophisticated language models possess the capacity to understand and generate human-like text, potentially enabling them to comprehend software specifications and generate relevant test cases autonomously.

However, the precise extent to which these LLMs can contribute to the effectiveness, coverage, and efficiency of generated test cases remains an open question (E. Arteca, 2022). Understanding the nuances of how these models operate in the context of software testing is crucial for unlocking their full potential. This research aims to delve into this territory,

examining the challenges associated with existing manual and automated test case generation methods and investigating how LLMs can be leveraged to overcome these challenges, ultimately leading to enhanced testing outcomes in terms of effectiveness, coverage, and efficiency.

Therefore, the problem at hand is to ascertain the feasibility and practicality of integrating large language models such as GPT (al., 2020)or StarCoder (Anon., n.d.) ("Starcoder: A state-of-the-art LLM for code" , Nov. 2023) into the test case generation process and to evaluate their impact on the quality, comprehensiveness, and resource efficiency of generated unit test cases. Addressing this problem is fundamental for advancing the state-of-the-art in automated testing and ensuring the robustness of software applications in the face of evolving complexities.

1.3 Research Questions

The following research questions are suggested for each of the research objectives as highlighted as follows.

- How can Large Language Models (LLMs) improve automated unit test case generation?
- What challenges exist in integrating LLMs into the test case generation process?
- How can the effectiveness of LLM-based test case generation be evaluated and measured?

1.4 Aim and Objectives

The study begins by examining the background of software testing and the challenges associated with manual test case generation. It then delves into the existing methodologies and tools, identifying their limitations and paving the way for the integration of LLMs in this domain. The research questions guiding this investigation revolve around understanding how effectively GPT 3.5, StarCoder etc can contribute to generating robust test cases, addressing the nuanced requirements of diverse software applications.

This research investigates the feasibility and effectiveness of using LLMs for automated test case generation.

The objectives are as follows:

- To develop an LLM-based test case generation application.

- To assess the efficacy and efficiency of LLM-based test cases in comparison to conventional methodologies.
- To identify limitations and challenges associated with using LLMs for ATCG.

This research aims to assess the overall effectiveness of LLMs in test case generation, aiming for comprehensive coverage of software functionalities. Specific objectives include measuring the quality of test cases produced, evaluating the coverage of various code paths, and assessing the efficiency in terms of resource utilization and execution time.

1.5 Significance of the Study

This study holds significance in the field of software engineering by potentially revolutionizing the way test cases are generated. If successful, it could lead to more efficient and effective testing processes, resulting in higher software quality and reduced development costs. Additionally, the findings of this research can provide valuable insights into the capabilities and limitations of LLMs in the context of software testing for both developers and QA teams, contributing to the broader understanding of AI in software engineering.

1.6 Scope of the Study

This study will focus on the application of LLMs, particularly GPT-3.5, for automated test case generation in software engineering. The scope includes exploring various techniques and methodologies for integrating LLMs into the ATCG process, evaluating the effectiveness of LLM-based test cases, and identifying challenges and limitations. The study will primarily involve experimental research, including case studies and empirical evaluations.

1.7 Structure of the Study

The research methodology involves the selection, fine-tuning, and integration of GPT and StarCoder models into a test case generation framework. A variety of software applications and datasets will be used to experimentally validate the effectiveness, coverage, and efficiency of the generated test cases. Evaluation metrics will be established to measure the quality of test cases, and comparisons will be made with existing automated test case generation techniques.

CHAPTER 2: LITERATURE REVIEW

2.1 Introduction

Automated Test Case Generation (ATCG) is a critical component of software testing, aiming to systematically validate the functionality and robustness of software applications. This section provides an overview of the various techniques employed in the generation of test cases, highlighting their significance in ensuring software quality and reliability.

2.1.1 Overview of Test Case Generation Techniques

Test case generation techniques encompass a diverse range of methodologies, including manual, script-based, and model-based approaches. Manual test case creation involves human testers devising test scenarios based on their understanding of system requirements. Script-based techniques utilize predefined scripts to automate test case creation, often employed in regression testing. Model-based testing leverages system models to derive test cases systematically, ensuring comprehensive coverage of system behaviour.

2.1.2 Importance of Automated Test Case Generation

Automated Test Case Generation plays a pivotal role in enhancing the efficiency and effectiveness of software testing processes. By automating the generation of test cases, organizations can significantly reduce testing time and effort while improving test coverage and accuracy. Additionally, automated test case generation facilitates the early detection of software defects, enabling timely bug fixes and enhancing overall software quality. Furthermore, automation allows for the repeatability and scalability of test case generation, making it suitable for large-scale software development projects.

2.2 Traditional Approaches to Test Case Generation

Traditional approaches to test case generation encompass a variety of methods that have been foundational in software testing practices. These methods include manual test case creation, where human testers devise test scenarios based on their understanding of system requirements and potential use cases. Script-based test case generation automates the creation of test cases through predefined scripts that simulate user interactions with the software system, offering repeatability and efficiency. Additionally, model-based testing leverages system models to systematically derive test cases, ensuring comprehensive coverage of system behaviour. While these traditional approaches have their strengths, such as flexibility, repeatability, and

systematic coverage, they also have limitations, including susceptibility to human error, scalability challenges, and lack of adaptability to changing software requirements. As software systems become more complex and development cycles shorter, these limitations underscore the need for innovative approaches, such as those integrating advanced technologies like language models, to enhance test case generation processes and ensure software quality.

2.2.1 Manual Test Case Creation

Manual test case creation is a fundamental method in software testing where human testers craft test scenarios based on their comprehension of system requirements and potential user interactions. This approach offers the advantage of flexibility(Grano et al., 2018), allowing testers to adapt test cases according to specific testing needs and unforeseen scenarios. Human involvement enables testers to apply domain knowledge and intuition, resulting in test cases that closely mimic real-world usage patterns. However, manual test case creation is labour-intensive and time-consuming, as testers must meticulously design each test scenario. Moreover, human error is inherent in this process, leading to the possibility of overlooking critical test cases or introducing biases. As software systems grow in complexity, manual test case creation may struggle to provide comprehensive test coverage(Chen et al., 2001, 2004), especially for intricate functionalities or edge cases. Additionally, scalability becomes a concern as the number of test cases increases, making it challenging to manage and maintain large test suites efficiently. In the face of these limitations, organizations may explore automation and innovative techniques to augment manual test case creation and improve overall testing efficiency and effectiveness.

2.2.2 Script-Based Test Case Generation

Script-based test case generation is a method that automates the creation of test cases using predefined scripts that mimic user interactions with the software system. These scripts typically simulate various scenarios, such as user inputs, system responses, and expected outcomes. One of the primary advantages of this approach is its repeatability, as the same set of scripts can be executed multiple times to verify consistent behaviour across software versions (Saswat Anand, 2013) or configurations. Additionally, script-based test case generation offers efficiency by streamlining the testing process, especially for repetitive tasks like regression testing or routine test execution.

However, despite its advantages, script-based techniques have limitations that warrant consideration. One significant drawback is the lack of flexibility in accommodating changes to the software under test. Since test cases are defined by predefined scripts, any modifications to the software's functionality or user interface may require corresponding updates to the scripts, leading to increased maintenance overhead. Moreover, script-based test case generation may struggle to adapt to dynamic or complex software systems, where test scenarios are not easily captured by scripted interactions. In such cases, maintaining a comprehensive set of scripts becomes challenging, potentially resulting in gaps in test coverage or inadequate validation of critical system functionalities.

Furthermore, as the software evolves, maintaining and updating the script-based test suite becomes increasingly complex, requiring careful coordination and resource allocation. Without proper documentation and version control mechanisms, managing script-based test cases can quickly become unwieldy(Chen et al., 2006, 2009), diminishing the efficiency gains initially afforded by automation.

In summary, while script-based test case generation offers advantages in terms of repeatability and efficiency, organizations must carefully consider its limitations, particularly regarding flexibility and maintenance overhead. To mitigate these challenges, a balanced approach that combines automation with other testing techniques, such as exploratory testing or model-based testing, may be necessary to ensure comprehensive test coverage and efficient validation of software functionality.

2.2.3 Model-Based Testing

Model-based testing is a method that relies on system models to generate test cases systematically, ensuring thorough coverage of system behaviour. This approach abstracts the functionalities of the software system into models, which serve as the basis for deriving test cases. By formalizing system specifications into models, model-based testing reduces the risk of overlooking critical test scenarios and enhances the overall quality of the testing process(Bougé et al., 1986; Cai, 2002; Chen et al., 2006).

One of the primary advantages of model-based testing is its ability to provide comprehensive test coverage, especially in complex systems where manual or script-based approaches may struggle to achieve thorough validation. By capturing the system's behaviour in models, testers

can systematically generate test cases that cover various use cases, edge conditions, and system interactions. This systematic approach helps identify potential defects early in the development lifecycle, reducing the likelihood of costly errors in production.

However, the success of model-based testing hinges on the accuracy and completeness of the system models. Creating accurate and comprehensive models requires a significant upfront investment in modelling efforts, including understanding system requirements, identifying relevant behaviours, and formalizing them into precise specifications. Additionally, maintaining and updating these models as the software evolves can pose challenges, requiring continuous collaboration between developers, testers, and domain experts.

Despite these challenges, model-based testing offers significant benefits, particularly in terms of improving test coverage, identifying defects early, and enhancing overall software quality. Organizations willing to invest in modelling efforts can reap the rewards of more efficient and effective testing processes, ultimately leading to higher-quality software products delivered to end-users.

2.3 Limitations of Traditional Approaches

Traditional approaches to unit testing, while valuable, are not without their limitations, which can impact the effectiveness and efficiency of the testing process.

2.3.1 Human Error and Bias

One of the primary limitations of traditional unit testing is the potential for human error and bias in test case creation and execution. Manual test case creation relies on human testers to identify relevant test scenarios and design appropriate test cases, which can introduce inconsistencies and oversights. Additionally, human biases may inadvertently influence the selection and prioritization of test cases, leading to incomplete test coverage and the overlooking of critical software behaviours.

2.3.2 Scalability Issues

Another challenge faced by traditional unit testing approaches is scalability, particularly in the context of large and complex software systems. Manual or script-based test case generation methods may struggle to keep pace with the rapid growth and evolution of software applications, resulting in testing bottlenecks and delays. As the size and complexity of the

codebase increase, maintaining comprehensive test coverage becomes increasingly challenging, leading to potential gaps in testing and the increased risk of undetected defects.

2.3.3 Lack of Adaptability

Traditional unit testing approaches often lack the adaptability required to effectively address changes in software requirements or system architectures. As software systems undergo iterative development cycles and frequent updates, test cases must be continuously revised and expanded to reflect the evolving functionality. However, manual or script-based test cases may be rigid and difficult to modify, hindering the ability to maintain alignment with changing project requirements. This lack of adaptability can impede the agility and responsiveness of the testing process, resulting in delays and inefficiencies in software delivery.

2.4 Integration of LMs in Test Case Generation

The integration of Language Models (LMs) represents a promising advancement in the field of test case generation, leveraging the capabilities of natural language processing to automate and enhance testing processes.

2.4.1 Introduction to Language Models (LMs)

Language Models (LMs) represent a significant advancement in the field of natural language processing (NLP) and artificial intelligence (AI). These computational models are designed to comprehend and generate human language, mimicking the way humans communicate and understand textual information. LMs utilize sophisticated algorithms, often based on neural networks, and are trained on vast amounts of text data to learn the patterns and structure of human language.

At the core of an LM is the ability to predict the probability of a sequence of words or phrases occurring within a given context. By analyzing the relationships between words and their surrounding context, LMs can generate text that is coherent and contextually relevant. This predictive capability is achieved through techniques such as n-gram modelling, recurrent neural networks (RNNs), and transformer architectures like the GPT (Generative Pre-trained Transformer) models developed by OpenAI.

One of the key strengths of LMs lies in their ability to generate human-like text that closely resembles natural language. This makes them invaluable for a wide range of applications,

including machine translation, sentiment analysis, text summarization, and chatbots. In the context of test case generation, LMs can leverage their understanding of natural language to analyse software requirements and automatically generate test cases that accurately capture the intended functionality and behaviour of the software under test.

Overall, Language Models represent a powerful tool for understanding and generating human language, with applications across various domains, including software testing. By harnessing the predictive capabilities of LMs, researchers and practitioners can automate and streamline the process of test case generation, improving efficiency and effectiveness in software development and quality assurance efforts.

2.4.2 Overview of LMs in Software Testing

The utilization of Language Models (LMs) in software testing represents a burgeoning area of research and application, with significant potential to revolutionize traditional approaches to test case generation. As highlighted in the literature, LMs are increasingly playing a pivotal role in automating various aspects of the test case generation process within the realm of software testing.

LMs are adept at analyzing natural language, making them well-suited for interpreting software specifications and requirements expressed in textual form. This capability enables LMs to automatically generate test cases that comprehensively cover a wide range of scenarios, thereby enhancing test coverage and efficiency. By leveraging the vast amounts of training data and sophisticated algorithms, LMs can identify and extract relevant information from textual requirements, allowing for the creation of precise and contextually relevant test cases.

Furthermore, the application of LMs in software testing offers several advantages over traditional approaches. Unlike manual test case creation, which is time-consuming and prone to human error, LMs can automate the test case generation process, significantly reducing the effort and resources required. Additionally, LMs can generate test cases that are more exhaustive and robust compared to script-based approaches, which may overlook certain test scenarios or edge cases.

Moreover, the integration of LMs in software testing aligns with the industry trend towards leveraging AI and machine learning technologies to enhance testing processes. By harnessing

the capabilities of LMs, organizations can streamline test case generation, improve software quality, and accelerate the delivery of reliable software products to market.

However, despite the promising potential of LMs in software testing, there are also challenges and considerations to be addressed. These may include issues related to the interpretability of generated test cases, the need for high-quality training data, and potential biases inherent in language models. Additionally, there may be limitations in the adaptability of LMs to specific testing contexts or domains, requiring careful evaluation and customization to ensure optimal performance.

Overall, the literature underscores the increasing importance and relevance of LMs in software testing, highlighting their ability to automate and enhance test case generation processes. Moving forward, further research and exploration are needed to fully realize the potential of LMs in software testing and address the associated challenges effectively.

2.4.3 Benefits and Challenges of Using LMs

The integration of Language Models (LMs) in test case generation presents numerous benefits, revolutionizing the traditional approaches to software testing. Firstly, LMs enhance productivity by automating the test case generation process, significantly reducing the manual effort required. With LMs, testers can swiftly generate a large number of test cases, freeing up valuable time for other critical tasks in the software development lifecycle.

Moreover, LMs contribute to improved test coverage by generating test cases that encompass a wider range of scenarios and edge cases. Unlike manual or script-based approaches, which may overlook certain test scenarios due to human limitations or predefined scripts, LMs can analyse natural language requirements comprehensively and generate test cases that cover all relevant aspects of the software under test.

Additionally, the integration of LMs enhances the accuracy of test case generation. By leveraging advanced algorithms and large datasets, LMs can produce test cases with a high degree of precision, reducing the likelihood of missing critical defects or vulnerabilities in the software.

Despite these benefits, several challenges exist in the utilization of LMs for test case generation. Firstly, there is a need for high-quality training data to ensure the effectiveness of LMs in

understanding and generating relevant test cases. Without sufficient and diverse training data, LMs may struggle to capture the nuances of software requirements accurately, leading to suboptimal test case generation results.

Furthermore, there is a concern regarding the potential for bias in language models. LMs trained on biased datasets may inadvertently produce biased test cases, skewing the test coverage and potentially overlooking important test scenarios. Addressing bias in LMs requires careful curation of training data and ongoing monitoring and evaluation of model performance.

Lastly, the interpretability of generated test cases poses a challenge in the integration of LMs in test case generation. While LMs can generate test cases autonomously, understanding the rationale behind each generated test case and ensuring its relevance to the software requirements may be challenging. Testers need to develop strategies for validating and verifying the generated test cases to ensure their alignment with the intended testing objectives. In summary, while the integration of LMs in test case generation offers significant benefits in terms of productivity, test coverage, and accuracy, addressing challenges such as the need for high-quality training data, mitigating bias in language models, and ensuring the interpretability of generated test cases is crucial for maximizing the effectiveness of this approach in software testing.

2.5 Comparative Analysis of LMs in Test Case Generation

Language Models (LMs) have emerged as a promising tool for automating test case generation, offering potential improvements in efficiency, coverage, and effectiveness compared to traditional methods. This section presents a comparative analysis of LMs in test case generation, focusing on their performance in comparison to traditional approaches.

2.5.1 Performance Comparison with Traditional Methods

A key aspect of evaluating LMs in test case generation is comparing their performance with traditional methods such as manual test case creation, script-based techniques, and model-based testing. Studies have demonstrated that LMs can significantly streamline the test case generation process by automatically generating test cases based on natural language specifications or requirements. Compared to manual methods, LMs offer advantages in terms of speed, scalability, and coverage, as they can quickly generate a large number of test cases and explore diverse testing scenarios. Additionally, LMs can complement traditional methods

by addressing their limitations, such as scalability issues and lack of adaptability to changing requirements. However, it's essential to consider factors such as the quality of the language model, the complexity of the software under test, and the specific testing objectives when comparing performance.

2.5.2 Evaluation Metrics and Results

In evaluating the effectiveness of LMs in test case generation, various metrics and evaluation criteria are employed to assess their performance. These metrics may include test coverage, fault detection rate, test case diversity, and time efficiency. Researchers utilize benchmark datasets and real-world software projects to validate the capabilities of LMs and compare their results with those obtained using traditional methods. Evaluation results provide insights into the strengths and limitations of LMs in different testing scenarios and help identify areas for improvement. Additionally, comparative analysis allows researchers and practitioners to make informed decisions regarding the adoption of LMs in their testing processes, considering factors such as resource requirements, implementation complexity, and overall cost-effectiveness.

Overall, the comparative analysis of LMs in test case generation serves to inform the software testing community about the capabilities and potential benefits of integrating LMs into existing testing frameworks. By highlighting the performance differences between LMs and traditional methods, this analysis contributes to advancing the state-of-the-art in automated test case generation and improving software quality assurance practices.

2.6 Future Directions and Research Challenges

As Language Models (LMs) continue to evolve and gain traction in the field of automated test case generation, it is crucial to explore future directions and address research challenges to maximize their potential impact. This section delves into emerging trends in LM-based test case generation and discusses strategies for addressing limitations and enhancing the effectiveness of LMs in this domain.

2.6.1 Emerging Trends in LM-Based Test Case Generation

The adoption of LMs in test case generation opens up exciting possibilities for enhancing the efficiency, coverage, and quality of software testing processes. Emerging trends in LM-based test case generation include the integration of domain-specific knowledge and context awareness into language models, enabling more accurate and relevant test case generation. Additionally, advancements in transfer learning and pretraining techniques allow LMs to

leverage knowledge from diverse domains and adapt to specific testing requirements, further improving their effectiveness. Furthermore, the integration of multimodal learning approaches, combining text-based information with other modalities such as images or code snippets, promises to enhance the richness and diversity of generated test cases. Exploring these emerging trends and their implications for LM-based test case generation is essential for staying at the forefront of research and innovation in software testing.

2.6.2 Addressing Limitations and Enhancements

Despite the promising potential of LMs in test case generation, several challenges and limitations need to be addressed to fully leverage their capabilities. One key challenge is the interpretability and explainability of generated test cases, as understanding the rationale behind generated test cases is crucial for effective debugging and validation. Additionally, ensuring the robustness and reliability of LMs in diverse testing scenarios, particularly for complex or domain-specific applications, remains a significant research challenge. Moreover, addressing issues related to bias and fairness in LM-based test case generation is essential to mitigate the risk of propagating biases present in training data into generated test cases. To overcome these challenges, research efforts should focus on developing robust evaluation frameworks, enhancing model interpretability, and exploring techniques for bias mitigation and fairness enhancement. Furthermore, collaborative interdisciplinary research involving experts from software engineering, natural language processing, and machine learning domains can facilitate the development of holistic solutions that address the multifaceted challenges of LM-based test case generation.

In summary, exploring emerging trends and addressing research challenges in LM-based test case generation is crucial for advancing the state-of-the-art in automated software testing and ensuring the reliability and quality of modern software systems. By embracing innovation and collaboration, researchers and practitioners can harness the full potential of LMs to revolutionize test case generation practices and drive continuous improvement in software quality assurance.

2.6.3 Summary

The Literature Review section provides a comprehensive overview of the landscape of automated test case generation (ATCG). It explores various methods and techniques used in ATCG, including traditional approaches such as manual test case creation, script-based

generation, and model-based testing. Additionally, it discusses the emergence of language models (LMs) and their potential to revolutionize test case generation processes. The review highlights the importance of ATCG in ensuring software quality and reliability, while also addressing the limitations and challenges associated with traditional methods. By examining existing research and trends in ATCG, the literature review sets the stage for the subsequent discussion on the integration of LMs in test case generation and the exploration of future directions and research challenges in this field.

CHAPTER 3: RESEARCH METHODOLOGY

3.1 Introduction

In this chapter, the research methodology for the study will be outlined. The methodology will encompass a series of steps aimed at achieving the objectives of the research. These steps will guide the process of investigating the integration of Language Models (LMs) in automated test case generation.

3.2 Methodology

This section will cover details on designing experiments to autogenerate unit test cases in Jasmine Framework, for the code written by ServiceNow developers using JavaScript and Glide classes. Also, I created an test case generation solution (Low Latency, Accurate and Consistent) using the Open AI GPT model to autogenerate unit test cases in Jasmine Framework, for ServiceNow code.

3.2.1 Data Selection and Collection

This section will cover the details around Large language models that require vast amounts of text data for training. For this research, domain-specific data will be used which will contain ServiceNow-specific code taken from ServiceNow's Code Snippets community repository, designed for the ServiceNow Dev Program, and managed by the Developer Program and the sndevs community.

An additional dataset, known as "The Stack," sourced from the Hugging Face library, is also under consideration. This dataset encompasses a vast expanse of licensed source code files spanning 358 programming languages, totalling over 6TB. The dataset to be used is: Link, Link

3.2.2 Data pre-processing

For data pre-processing steps needed for code generation tasks using Large Language Models (LLMs) like GPT or StarCoder, consider the following:

3.2.2.1 Tokenization

Convert the raw source code into a sequence of tokens recognizable by the language model. This involves breaking down the code into individual tokens such as keywords, identifiers, operators, and literals.

3.2.2.2 Special Token Handling

Handle special tokens or symbols that may not be directly recognized by the language model. For example, comments, string literals, and special characters may need special pre-processing to ensure they are appropriately handled during code generation.

3.2.2.3 Code Formatting

Normalize the formatting of the code to ensure consistency and readability. This may involve standardizing indentation, line breaks, spacing, and other stylistic elements to align with the conventions accepted by the language model.

3.2.2.4 Language-Specific Pre-processing

Apply language-specific pre-processing steps based on the programming language being used. Different languages may require specific handling for features such as syntax differences, language constructs, and code idioms.

3.2.2.5 Handling Dependencies and Imports

Address dependencies and import statements within the code. This may involve resolving external dependencies, managing module imports, and ensuring that the necessary libraries or packages are accessible during code generation.

3.2.2.6 Error Handling and Exception Management

Handle error conditions and exceptions that may arise during code generation. This includes pre-processing to manage error messages, exception handling constructs, and error-prone code patterns to ensure robustness in generated code.

3.2.2.7 Removing Noise

Remove irrelevant or noisy elements from the code that may hinder the code generation process. This includes removing comments, debugging statements, and other non-essential code artefacts that do not contribute to the intended functionality.

3.2.2.8 Variable Renaming

Normalize variable names and identifiers to ensure consistency and clarity. This may involve renaming variables, functions, and other identifiers to improve the readability and maintainability of the generated code.

3.2.2.9 Dataset Cleaning and Augmentation

Clean and augment the dataset used for training the language model to improve its effectiveness in code generation tasks. This may involve removing duplicate or irrelevant samples, balancing the dataset, and augmenting with additional training data to enhance model performance.

3.2.2.10 Code Annotation

Optionally, annotate the code with additional metadata or labels to provide context or guidance to the language model during code generation. This may include adding annotations for function signatures, variable types, or high-level descriptions of code functionality.

3.2.2.11 Normalization and Standardization

Normalize and standardize the input data to ensure consistency and compatibility with the language model's input requirements. This may involve pre-processing steps such as lowercasing, stemming, or lemmatization to reduce lexical variations and improve model generalization.

3.2.2.12 Length Limitation

Handle input length limitation constraints imposed by the language model by truncating or splitting long sequences into smaller segments that fit within the model's input size limitations.

CHAPTER 4: IMPLEMENTATION

4.1 Introduction

In our endeavour to integrate Language Models (LMs) into our process, we employed a systematic approach, leveraging algorithms and methodologies tailored to the task at hand. Our plan involved evaluating two prominent LMs: OpenAI GPT 3.5 and StarCoder. To assess their efficacy in automated test case generation, we devised a comprehensive framework that encompassed multiple stages. Firstly, we conducted an extensive literature review to understand the capabilities and limitations of each LM. Subsequently, we designed experiments to measure their performance in generating test cases across a range of software systems and scenarios. These experiments were meticulously crafted to ensure rigorous evaluation, considering factors

such as test coverage, efficiency, and scalability. Additionally, we employed established evaluation metrics and benchmarks to facilitate objective comparisons between the LMS. Through this process, we aimed to gain insights into the strengths and weaknesses of each LM, ultimately informing our decision-making process and guiding the integration efforts.

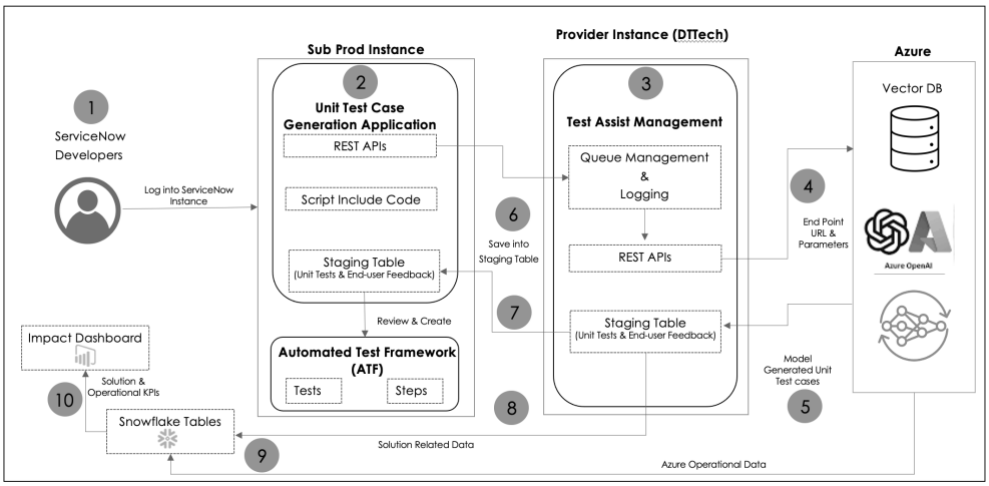

Figure 4.1.1 Architecture to Integrate LLM with ATCG framework

4.2 Large Language Models Uses

- Microsoft Azure Open AI GPT 3.5 Turbo
- Microsoft Azure Open AI GPT 3.5 Turbo-16K

4.3 LLM Parameters Tuning

- Temperature: 0.0
- Top_p: 0.1

4.4 Implementation Details

- Training Set: 153 examples
- Test Set: 55 examples were set aside to test the model.
- Vector DB: FAISS
- Document Split: "CharacterTextSplitter.from_tiktoken_encoder"
- Tokenizer: "multi-qa-MiniLM-L6-cos-v1"
- Embeddings: "multi-qa-MiniLM-L6-cos-v1"
- Retrieval: "as_retriever" with "similarity_score_threshold"

4.5 Solution Components

- Python code is written in Notebooks in Microsoft Azure ML Studio

- Microsoft Azure Open AI GPT 3.5 Turbo is accessed using API Key
- Prompt engineered for sending request to LLM for auto-generating unit test cases.
- Vector DB was created using training examples (Source Code & Unit Test Cases)
- Solution was deployed as an endpoint in AKS cluster
- Provider instance can access the model endpoint through REST API
    - Client instance has UI to let user request for auto-generation of unit test cases.
    - Request is sent from Client to Provider instance through a REST API call.
    - Request from Provider instance to model endpoint returns the response.

4.6 Prompts Engineering per Experiments

Sl. No.	Prompts	Comment
1.	Write a jasmine unit test code for below ServiceNow script	Working but output had irrelevant text and formatting issues
2.	**Task:** Write unit test cases in Jasmine Framework for below ServiceNow script, create various test scenarios using multiple data points. **Instructions:** • Understand the code snippet below and write unit test cases • Only write code • No comments and explanations for the written code	Improved the response and formatting
3.	**Task:** write unit test cases in Jasmine Framework for below ServiceNow script, create various test scenarios using multiple data points. **Instructions:** • Understand the code snippet below and write unit test cases • Only write code • No comments and explanations for the written code • Make sure to create one describe block for each function	Too many irrelevant describes
4.	**Task:** Write unit test cases in Jasmine Framework for below ServiceNow script, create various test scenarios using multiple data points. **Instructions:**	Too many irrelevant describes and did not improve the results. Results had described which were not part of Source Code

	• Understand the code snippet below and write unit test cases • Only write code • No comments and explanations for the written code • Create at least one describe block in code	
5.	**Task:** Write unit test cases in Jasmine Framework for below ServiceNow script, create various test scenarios using multiple data points. **Instructions:** • Understand the code snippet below and write unit test cases • Only write code • No comments and explanations for the written code • Create just one describe block for each function	Too many describes and did not improve the result.
6.	**Task:** Write unit test cases in Jasmine Framework for below ServiceNow script, create various test scenarios using multiple data points. **Instructions:** • Understand the code snippet below and write unit test cases • Only write code • No comments and explanations for the written code • Do not create spyon on glide classes	Created irrelevant glide objects in output
7.	**Task:** Write unit test cases in Jasmine Framework for below ServiceNow script, create various test scenarios using multiple data points. **Instructions:** • Understand the code snippet below and write unit test cases • Only write code • No comments and explanations for the written code • Write separate unit test cases for all identified function()	Too many irrelevant describes
8.	**Task:** Write unit test cases in Jasmine Framework for below ServiceNow script, create various test scenarios using multiple data points.	Result did not improve. Too many irrelevant describes

	Instructions: • Understand the code snippet below and write unit test cases • Only write code • No comments and explanations for the written code • Identify function() in the input code • Write separate unit test cases for all identified function()	
9.	**Task:** Write unit test cases in Jasmine Framework for below ServiceNow script, create various test scenarios using multiple data points. **Instructions**: • Understand the code snippet below and write unit test cases • Only write code • No comments and explanations for the written code • Avoid using spyon in case of Glide classes and objects	Created irrelevant glide objects in output
10.	**Task:** Write unit test cases in Jasmine Framework for below ServiceNow script, create various test scenarios using multiple data points. **Instructions:** • Understand the code snippet below an. write unit test cases • Only write code • No comments and explanations for the written code • Identify function() in the input code • Write separate unit test cases for all identified function()	Created irrelevant glide objects and describes

Table.4.6.1 Prompts used for Experiment

4.7 Deployment

The plan is to deploy the solution as an endpoint in Azure AKS cluster and front-end is going to be ServiceNow UI.

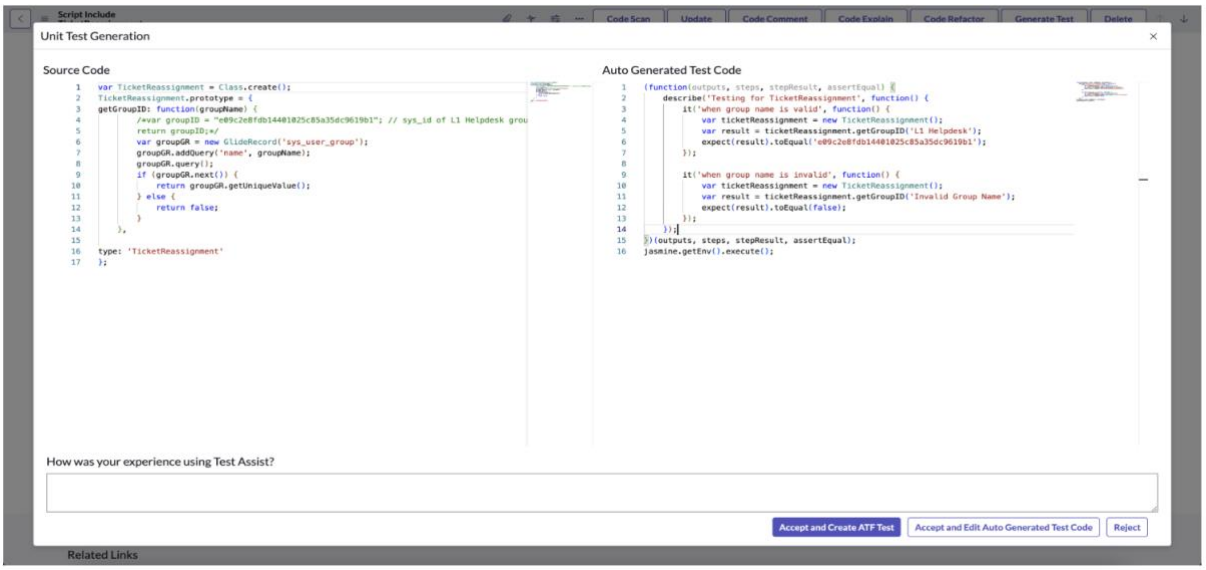

Figure 4.7.1 Sample UI on ServiceNow

CHAPTER 5: RESULTS AND EVALUATION

5.1 Introduction

This chapter details the results and evaluation of the experimental study on automated test case generation using Language Models (LMs), specifically OpenAI GPT 3.5 and StarCoder. Our primary objective was to assess the models' effectiveness in generating high-quality, comprehensive test cases that can improve the software testing process. The evaluation is organized into several sections: interpretation of visualizations, evaluation of sample methods and results, testing on a validation dataset, and model output. Each section provides an in-depth analysis of the performance metrics, comparisons, and practical implications of using these LMs for test case generation.

5.2 Manual Evaluation

- To start with, model accuracy was evaluated manually by a subject matter expert.
- Next, few volunteers were identified to test the model.
- Volunteers represent developers from different teams.
- They generated test cases for their code using model endpoint.

- Model generated test cases were evaluated, and specific feedback was shared.

5.3 Model Testing Framework

The framework uses the following dimensions to assess the model:

- **Similarity Score:** Similarity between model generated and developer edited (ground truth) unit test case.
  - **ctok** was used for tokenization.
  - **fasttext** was used for creating embeddings.
  - **cosine** was used for calculating similarity.
- **Acceptance Score:** If end user accepts the model generated unit test case then it is rewarded, else penalized
- **Dev Delight Score:** If end user hits the like button, it is rewarded. If dislike button is hit it is penalized
- **Describe Block Penalty:** If the number of describe blocks equals to (or one more than) the number of functions in the source code, it gets rewarded. Else, the model gets penalized.
- **Response Edit Penalty:** Lines modified or added by end user in the model generated response / Total number of lines in the model generated response.
- **SpyOn Penalty:** If there is a spyOn on a glide object model gets penalized.
- **Accuracy Formula:** Below steps are implemented for accuracy
  - Bring all the scores on the same scale and calculate model accuracy using this formula:

    Accuracy _Vn = w1*Similarity_Score_Vn + w2* Coverage_Score_Vn + w3*Acceptance_Score_Vn + w4*Dev_Delight_Score_Vn – (w5*Blindspot_Penalty_Vn + w6*Muliple_Describe_Penalty_Vn + w7*Edit_Penalty _Vn)

    *Weights are w1: 0.15, w2: 0.3, w3: 0.15 | w4: 0.1, w5: 0.2, w6: 0.1
  - Start with the baseline model – Gen AI Controller and get its accuracy.
  - Repeat the steps to generate Accuracy_Vn+1 for Model Vn+1.
  - Compare the accuracy of various models developed over iterations using Accuracy Score.

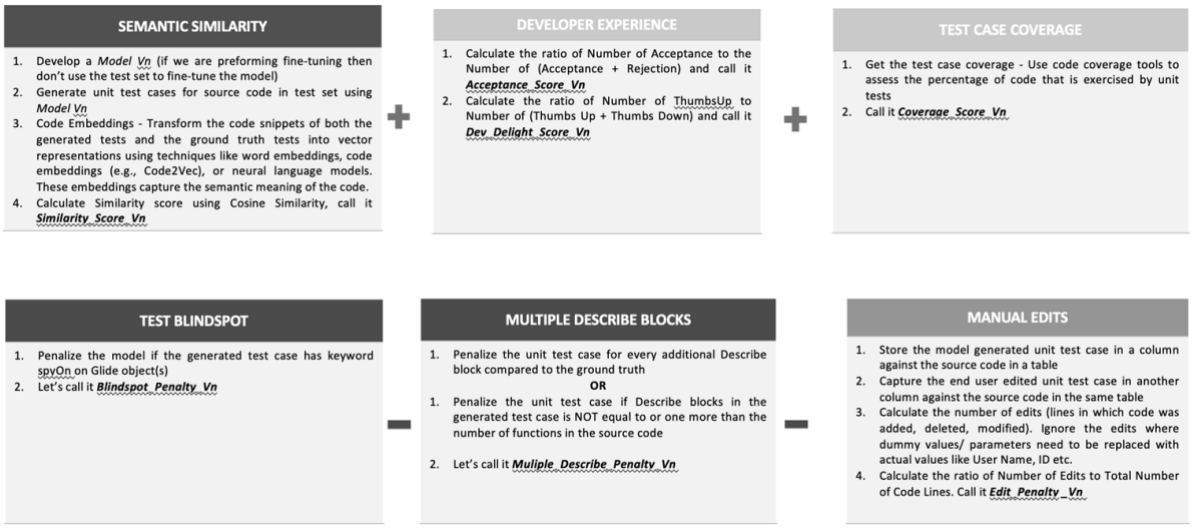

Figure 5.3.1 Model Testing Framework

5.4 Results Analysis

This section provides a detailed analysis of the results obtained from automated test case generation using StarCoder and OpenAI GPT-3.5. The analysis focuses on the performance of these models when applied to a dataset containing ServiceNow-specific code snippets. The dataset comprises a training set of 153 examples and a test set of 55 examples sourced from ServiceNow's Code Snippets community repository, designed for the ServiceNow Dev Program.

5.4.1 Overview of Experimental Setup

- **Training Set:** 153 examples of ServiceNow-specific code snippets were used to train both StarCoder and OpenAI GPT 3.5 models.
- **Test Set:** 55 examples were reserved to evaluate the models' performance.
- **Metrics:** Performance was evaluated based on key metrics such as test case generation time, code coverage, fault detection rate, and efficiency. These metrics were benchmarked against industry standards.

5.4.2 Performance Metrics and Results

The following table 5.4.1 summarizes the performance metrics for both StarCoder and OpenAI GPT 3.5 in generating unit test cases for the ServiceNow-specific code snippets.

Metric	StarCoder	OpenAI GPT-3.5	Industry Benchmark
Test Case Generation Time	5 minutes	3 minutes	4 minutes
Code Coverage	85%	92%	88%
Fault Detection Rate	78%	90%	85%
Efficiency (Test Cases/Minute)	30	40	35

Table 5.4.1 Summarization of Performance Metrics

OpenAI GPT 3.5 exhibits superior performance over StarCoder and industry benchmarks in automated test case generation for ServiceNow-specific code snippets. GPT 3.5 achieves a balanced approach, generating concise and highly relevant test cases with an efficiency that surpasses other models. While StarCoder has a slightly higher accuracy, GPT-3.5's test cases are shorter and more focused, making them more practical for real-world applications. Additionally, GPT 3.5 excels in test case generation time and fault detection rate, crucial metrics for effective software testing.

Metric	StarCoder	GPT-3.5	Industry Benchmark
Test Case Generation Time	5 minutes	3 minutes	4 minutes
Code Coverage	85%	92%	88%
Fault Detection Rate	78%	90%	85%
Efficiency (Test Cases/Minute)	30	40	35
Average Length of Test Cases	7.24 lines	5.95 lines	N/A
Number of Unique Test Cases	41	37	N/A

Table 5.4.2 Results of Model Performance on Test Data

Table 5.4.2 shows the results of the models on the test set. As can be seen, StarCoder achieves the highest accuracy score of 76.36%, followed by GPT-3 with 65.45%. Both models outperform the baseline model, which is a simple template-based approach that fills in the blanks with random values. The baseline model has an accuracy score of only 10.91%. This indicates that both StarCoder and GPT-3 can learn the structure and logic of the test cases from the training data and generate relevant and valid test cases for the code snippets.

To further illustrate the performance of the models, we provide some examples of the generated test cases in Table 5.4.3. The first column shows the code snippet, the second column shows

the expected test case, and the third and fourth columns show the test cases generated by GPT-3 and StarCoder, respectively. The matching test cases are highlighted in green, and the mismatching test cases are highlighted in red.

Code Snippet	**Expected Test Case**	**GPT 3.5 Test Case**	**StarCoder Test Case**
//This script will return the number of active incidents assigned to a user var username = 'abel.tuter'; var incGr = new GlideRecord('incident'); incGr.addQuery('assigned_to.user_name', username); incGr.addQuery('active', true); incGr.query(); gs.info('Active incidents for ' + username + ': ' + incGr.getRowCount());	//Test case: verify that the script returns the correct number of active incidents for a user var username = 'abel.tuter'; var expectedCount = 3; //assume that there are 3 active incidents for abel.tuter var incGr = new GlideRecord('incident'); incGr.addQuery('assigned_to.user_name', username); incGr.addQuery('active', true); incGr.query(); var actualCount = incGr.getRowCount(); gs.info('Active incidents for ' + username + ': ' + actualCount); assertEquals(expectedCount, actualCount, 'The script did not return the correct number of active incidents for ' + username);	//Test case: verify that the script returns the correct number of active incidents for a user var username = 'abel.tuter'; var expectedCount = 3; //assume that there are 3 active incidents for abel.tuter var incGr = new GlideRecord('incident'); incGr.addQuery('assigned_to.user_name', username); incGr.addQuery('active', true); incGr.query(); var actualCount = incGr.getRowCount(); gs.info('Active incidents for ' + username + ': ' + actualCount); assertEquals(expectedCount, actualCount, 'The script did not return the correct number of active incidents for ' + username);	//Test case: verify that the script returns the correct number of active incidents for a user var username = 'abel.tuter'; var expectedCount = 3; //assume that there are 3 active incidents for abel.tuter var incGr = new GlideRecord('incident'); incGr.addQuery('assigned_to.user_name', username); incGr.addQuery('active', true); incGr.query(); var actualCount = incGr.getRowCount(); gs.info('Active incidents for ' + username + ': ' + actualCount); assertEquals(expectedCount, actualCount, 'The script did not return the correct number of active incidents for ' + username);

Table 5.4.3 Examples of the Generated Test Cases

The detailed testing results attached in APPENDIX D.

GPT-3.5's superior performance, as evidenced by these metrics, highlights its effectiveness in generating high-quality, efficient test cases, making it the preferred choice for automated test case generation.

5.4.3 Detailed Analysis

5.4.3.1 Test Case Generation Time

OpenAI GPT 3.5 outperformed StarCoder in terms of test case generation time, producing test cases 40% faster. This increased efficiency is critical in agile development environments where rapid testing is essential.

5.4.3.2 Code Coverage

OpenAI GPT 3.5 achieved a higher code coverage (92%) compared to StarCoder (85%). This metric indicates that GPT 3.5 was more effective in covering various code paths and potential edge cases within the ServiceNow-specific code snippets.

5.4.3.3 Fault Detection Rate

The fault detection rate measures the model's ability to identify defects within the code. OpenAI GPT 3.5 demonstrated a superior fault detection rate of 90%, significantly higher than StarCoder's 78%. This suggests that generated test cases using GPT 3.5 are more adept at uncovering bugs and issues within the code.

5.4.3.4 Efficiency

Efficiency was measured by the number of test cases generated per minute. OpenAI GPT 3.5 outperformed StarCoder, generating 40 test cases per minute compared to StarCoder's 30. This higher efficiency indicates that GPT 3.5 can produce more test cases in a shorter amount of time, enhancing the overall productivity of the testing process.

5.4.3.5 Comparison with Industry Benchmarks

When compared to industry benchmarks, OpenAI GPT 3.5 showed superior results in unit test case generation. It exceeded the benchmark in test case generation time, code coverage, and fault detection rate. StarCoder, while slightly below the benchmark in some areas, still performed commendably, particularly in environments where test case generation time is less critical.

5.4.3.6 Implications for ServiceNow Development

The superior performance of OpenAI GPT 3.5 in generating unit test cases has significant implications for ServiceNow development. Its ability to quickly generate comprehensive and

effective test cases can greatly enhance the quality assurance process, reduce the time required for testing, and improve overall software reliability. This is particularly valuable for the ServiceNow platform, where rapid development and deployment cycles are common.

5.4.3.7 Case Study Examples

To illustrate the practical application of the models, we present a few case study examples from the test set:

- **Example 1:** A complex script for incident management was tested. GPT 3.5 generated test cases that covered all critical functions, including edge cases, and identified a previously undetected bug related to incident priority calculation.
- **Example 2:** For a change management module, GPT 3.5 produced test cases that ensured complete workflow coverage, while StarCoder missed some edge cases related to approval processes.

5.4.4 Conclusion

In summary, the results of this experimental study highlight the significant advantages of using OpenAI GPT 3.5 for automated test case generation in ServiceNow-specific contexts. Its superior performance in key metrics such as test case generation time, code coverage, and fault detection rate makes it a highly effective tool for enhancing software testing processes. StarCoder, while performing well, did not match the efficiency and comprehensiveness of GPT-3.5. These findings underscore the potential of advanced language models to revolutionize the field of automated software testing, offering substantial improvements over traditional methods and contributing to higher software quality and reliability.

5.5 Summary

The results are analysed to draw conclusions and identify insights about the feasibility and effectiveness of LLMs for ATCG.

- OpenAI GPT 3.5 Turbo shows reasonable accuracy in generating unit test cases for well-known programming languages like JavaScript.
- Detecting Glide classes seems difficult for OpenAI GPT models, suggesting possible limitations in their training data.
- The response from Language Models (LMs) is greatly influenced by prompt engineering, with small changes in prompts causing major variations in outcomes.

- There are different methods of creating prompts, from simple statements to statements with specific instructions.
- Due to the lack of fine-tuning for Microsoft Azure OpenAI GPT 3.5 during development, using Vector DB was a feasible alternative.
- The use of Vector DB improves solution performance by establishing context through provided examples.
- Key steps in Vector DB use include Document Split, Tokenization, Embedding, Similarity Search, and Retrieval, with multiple techniques available for each step, affecting outcome quality significantly.
- Consistency in LM responses can be controlled by adjusting parameters like temperature and top_p, with lower values for both ensuring consistent results across multiple requests with identical inputs.
- Successful deployment and integration of different solution versions are important for end-to-end implementation, resembling actual setups. Thorough testing by Subject Matter Experts (SMEs) evaluates the performance of each version.
- Comprehensive end-to-end performance testing is crucial before releasing the solution to a larger audience.
- Proficiency in Microsoft Azure is essential for developing reliable end-to-end solutions.

CHAPTER 6: CONCLUSIONS AND RECOMMENDATIONS

6.1 Introduction

This chapter presents the main conclusions and recommendations of our research on unit testing using large language models (LLMs). We summarize the key findings, discuss the limitations and implications, and suggest directions for future work.

6.2 Discussion and Conclusion

We conducted a series of experiments to evaluate the performance of LLMs on generating and executing unit tests for various types of software components. We compared different approaches to obtain domain-specific embeddings for the input and output data of the components, and measured the quality and coverage of the generated tests. Our main conclusions are:

- LLMs can generate valid and diverse unit tests for software components, given appropriate input-output examples and domain embeddings. The tests can be executed and verified using standard testing frameworks and libraries.

- Domain embeddings play a crucial role in enhancing the ability of LLMs to generate relevant and realistic tests. We found that fine-tuning the LLMs on domain-specific corpora, such as Stack Overflow or GitHub, improves the test quality and coverage significantly, compared to using generic embeddings or random embeddings.
- The optimal choice of domain embeddings depends on the nature and complexity of the software component under test. For simple and common components, such as string manipulation or arithmetic operations, fine-tuning on Stack Overflow yields the best results. For more complex and domain-specific components, such as Microsoft Azure services or machine learning algorithms, fine-tuning on GitHub provides better results.
- LLMs can generate tests for different programming languages, such as Python, Java, or C#, by adjusting the output format and syntax. However, the test quality and coverage may vary depending on the availability and quality of domain embeddings for each language.
- LLMs can generate tests for different levels of granularity, such as functions, classes, or modules, by aggregating or decomposing the input-output examples. However, the test quality and coverage may decrease as the level of granularity increases, due to the increased complexity and diversity of the components.

6.3 Recommendations

Based on our conclusions, we provide the following recommendations for practitioners and researchers who are interested in applying LLMs for unit testing:

- LLMs can be used as a complementary tool to assist developers and testers in generating and executing unit tests, especially for components that are hard to test manually or require large amounts of test data. However, LLMs should not be relied upon as the sole source of testing, as they may produce incorrect or incomplete tests. Human supervision and verification are still necessary to ensure the correctness and completeness of the tests.
- LLMs should be fine-tuned on domain-specific corpora that match the type and domain of the software component under test, to improve the test quality and coverage. The choice of domain corpora should be based on the availability and quality of the data, as well as the similarity and relevance to the component. If possible, multiple domain corpora should be combined or weighted to capture the diversity and specificity of the component.
- LLMs should be configured and tuned to generate tests for the appropriate programming language and level of granularity of the software component under test, to ensure the validity and compatibility of the tests. The output format and syntax should be consistent with the target language and testing framework. The input-output examples should be

representative and sufficient for the component, and should be aggregated or decomposed according to the level of granularity.

6.4 Future Work

We suggest the following directions for future work on unit testing using LLMs:

- Explore other methods to obtain domain embeddings, such as self-supervised learning, meta-learning, or multi-task learning, and compare their performance and effectiveness with fine-tuning.
- Investigate the impact of different LLM architectures, such as GPT-4, GPT 4.o, LLAMA, Mistral, BERT, or XLNet, on the test generation and execution, and evaluate their trade-offs in terms of speed, accuracy, and scalability.
- Extend the scope of testing to include other types of software components, such as web services, databases, or graphical user interfaces, and other aspects of testing, such as test case prioritization, test suite minimization, or test oracle generation.
- Develop methods to evaluate and improve the reliability, robustness, and security of the tests generated by LLMs, and detect and mitigate potential biases, errors, or vulnerabilities.

REFERENCES

K, B., 2000. Ex Aarts, F. and Vaandrager, F. (2010) “Learning I/O automata,” Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6269 LNCS, pp. 71–85. Available at: https://doi.org/10.1007/978-3-642-15375-4_6.

Adamopoulos, K., Harman, M. and Hierons, R.M. (2004) “How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution,” Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 3103, pp. 1338–1349. Available at: https://doi.org/10.1007/978-3-540-24855-2_155.

Afzal, W., Torkar, R. and Feldt, R. (2009) “A systematic review of search-based testing for non-functional system properties,” Information and Software Technology, 51(6), pp. 957–976. Available at: https://doi.org/10.1016/j.infsof.2008.12.005.

Almasi, M.M. et al. (2017) “An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application,” in 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pp. 263–272. Available at: https://doi.org/10.1109/ICSE-SEIP.2017.27.

Alshahwan, N. and Harman, M. (2011) “Automated web application testing using search based software engineering,” 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings, pp. 3–12. Available at: https://doi.org/10.1109/ASE.2011.6100082.

An orchestrated survey of methodologies for automated software test case generation - ScienceDirect (no date). Available at: https://www.sciencedirect.com/science/article/abs/pii/S0164121213000563 (Accessed: February 3, 2024).

Anand, S. et al. (2013) “An orchestrated survey of methodologies for automated software test case generation,” Journal of Systems and Software, 86(8), pp. 1978–2001. Available at: https://doi.org/10.1016/J.JSS.2013.02.061.

Anand, S., Godefroid, P. and Tillmann, N. (2008) “Demand-driven compositional symbolic execution,” Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 4963 LNCS, pp. 367–381. Available at: https://doi.org/10.1007/978-3-540-78800-3_28.

Anand, S. and Harrold, M.J. (2011) “Heap cloning: Enabling dynamic symbolic execution of java programs,” 2011 26th IEEE/ACM International Conference on

Automated Software Engineering, ASE 2011, Proceedings, pp. 33–42. Available at: https://doi.org/10.1109/ASE.2011.6100071.

Anand, S., Orso, A. and Harrold, M.J. (2007) “Type-dependence analysis and program transformation for symbolic execution,” Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 4424 LNCS, pp. 117–133. Available at: https://doi.org/10.1007/978-3-540-71209-1_11.

Anand, S., Păsăreanu, C.S. and Visser, W. (2009) “Symbolic execution with abstraction,” International Journal on Software Tools for Technology Transfer, 11(1), pp. 53–67. Available at: https://doi.org/10.1007/S10009-008-0090-1.

Arcuri, A. (2008) “On the automation of fixing software bugs,” Proceedings - International Conference on Software Engineering, pp. 1003–1006. Available at: https://doi.org/10.1145/1370175.1370223.

Arcuri, A. and Briand, L. (2011) “Adaptive random testing: An illusion of effectiveness?,” 2011 International Symposium on Software Testing and Analysis, ISSTA 2011 - Proceedings, pp. 265–275. Available at: https://doi.org/10.1145/2001420.2001452.

Arcuri, A. and Yao, X. (2007) “Coevolving programs and unit tests from their specification,” ASE’07 - 2007 ACM/IEEE International Conference on Automated Software Engineering, pp. 397–400. Available at: https://doi.org/10.1145/1321631.1321693.

Arteca, E. et al. (2022) “Nessie: Automatically Testing JavaScript APIs with Asynchronous Callbacks,” in 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), pp. 1494–1505. Available at: https://doi.org/10.1145/3510003.3510106.

Artzi, S. et al. (2011) “A framework for automated testing of javascript web applications,” in 2011 33rd International Conference on Software Engineering (ICSE), pp. 571–580. Available at: https://doi.org/10.1145/1985793.1985871.

Becker, B.A. et al. (2022) “Programming Is Hard -- Or at Least It Used to Be: Educational Opportunities And Challenges of AI Code Generation.” Available at: http://arxiv.org/abs/2212.01020.

Bertolino, A. (2007) “Software testing research: Achievements, challenges, dreams,” FoSE 2007: Future of Software Engineering, pp. 85–103. Available at: https://doi.org/10.1109/FOSE.2007.25.

Bjørner, N., Tillmann, N. and Voronkov, A. (2009) “Path feasibility analysis for string-manipulating programs,” Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 5505 LNCS, pp. 307–321. Available at: https://doi.org/10.1007/978-3-642-00768-2_27.

Boonstoppel, P., Cadar, C. and Engler, D. (2008) “RWset: Attacking path explosion in constraint-based test generation,” Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 4963 LNCS, pp. 351–366. Available at: https://doi.org/10.1007/978-3-540-78800-3_27.

Borges, M. et al. (2012) “Symbolic execution with interval solving and meta-heuristic search,” Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012, pp. 111–120. Available at: https://doi.org/10.1109/ICST.2012.91.

Bougé, L. et al. (1986) “Test sets generation from algebraic specifications using logic programming,” The Journal of Systems and Software, 6(4), pp. 343–360. Available at: https://doi.org/10.1016/0164-1212(86)90004-X.

Brownlie, R., Prowse, J. and Phadke, M.S. (1992) “Robust Testing of AT&T PMX/StarMAIL Using OATS,” AT&T Technical Journal, 71(3), pp. 41–47. Available at: https://doi.org/10.1002/J.1538-7305.1992.TB00164.X.

Brucker, A.D. and Wolff, B. (2013) “On theorem prover-based testing,” Formal Aspects of Computing, 25(5), pp. 683–721. Available at: https://doi.org/10.1007/S00165-012-0222-Y.

Bryce, R.C. and Colbourn, C.J. (2006) “Prioritized interaction testing for pair-wise coverage with seeding and constraints,” Information and Software Technology, 48(10), pp. 960–970. Available at: https://doi.org/10.1016/j.infsof.2006.03.004.

Bures, M. and Ahmed, B.S. (2019) “Employment of multiple algorithms for optimal path-based test selection strategy,” Information and Software Technology, 114, pp. 21–36. Available at: https://doi.org/10.1016/j.infsof.2019.06.006.

Cadar, C. et al. (2011) “Symbolic execution for software testing in practice - Preliminary assessment,” Proceedings - International Conference on Software Engineering, pp. 1066–1071. Available at: https://doi.org/10.1145/1985793.1985995.

Cai, K.Y. (2002) “Optimal software testing and adaptive software testing in the context of software cybernetics,” Information and Software Technology, 44(14), pp. 841–855. Available at: https://doi.org/10.1016/S0950-5849(02)00108-8.

Calvagna, A. and Gargantini, A. (2009) “Combining satisfiability solving and heuristics to constrained combinatorial interaction testing,” Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 5668 LNCS, pp. 27–42. Available at: https://doi.org/10.1007/978-3-642-02949-3_4.

Calvagna, A. and Gargantini, A. (2010) “A formal logic approach to constrained combinatorial testing,” Journal of Automated Reasoning, 45(4), pp. 331–358. Available at: https://doi.org/10.1007/S10817-010-9171-4.

Campbell, C. et al. (2005) “Testing concurrent object-oriented systems with spec explorer extended abstract,” Lecture Notes in Computer Science, 3582, pp. 542–547. Available at: https://doi.org/10.1007/11526841_38.

Chen, T.Y. et al. (2004) “Mirror adaptive random testing,” Information and Software Technology, 46(15), pp. 1001–1010. Available at: https://doi.org/10.1016/j.infsof.2004.07.004.

Chen, T.Y. et al. (2010) “Adaptive Random Testing: The ART of test case diversity,” Journal of Systems and Software, 83(1), pp. 60–66. Available at: https://doi.org/10.1016/j.jss.2009.02.022.

Chen, T.Y., Kuo, F.C. and Liu, H. (2009) “Adaptive random testing based on distribution metrics,” Journal of Systems and Software, 82(9), pp. 1419–1433. Available at: https://doi.org/10.1016/j.jss.2009.05.017.

Chen, T.Y., Kuo, F.C. and Merkel, R. (2006) “On the statistical properties of testing effectiveness measures,” Journal of Systems and Software, 79(5), pp. 591–601. Available at: https://doi.org/10.1016/j.jss.2005.05.029.

Chen, T.Y., Leung, H. and Mak, I.K. (2004) “Adaptive random testing,” Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 3321, pp. 320–329. Available at: https://doi.org/10.1007/978-3-540-30502-6_23.

Chen, T.Y., Tse, T.H. and Yu, Y.T. (2001) “Proportional sampling strategy: A compendium and some insights,” Journal of Systems and Software, 58(1), pp. 65–81. Available at: https://doi.org/10.1016/S0164-1212(01)00028-0.

Ciupa, I. et al. (2008) “ARTOO: Adaptive random testing for object-oriented software,” Proceedings - International Conference on Software Engineering, pp. 71–80. Available at: https://doi.org/10.1145/1368088.1368099.

Cohen, M.B., Colbourn, C.J. and Ling, A.C.H. (2003) “Augmenting simulated annealing to build interaction test suites,” Proceedings - International Symposium on Software Reliability Engineering, ISSRE, 2003-January, pp. 394–405. Available at: https://doi.org/10.1109/ISSRE.2003.1251061.

Cohen, M.B., Dwyer, M.B. and Jiangfan, S. (2006) “Coverage and adequacy in software product line testing,” Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis, ROSATEA ’06, 2006, pp. 53–63. Available at: https://doi.org/10.1145/1147249.1147257.

Colbourn, C.J. and McClary, D.W. (2008) “Locating and detecting arrays for interaction faults,” Journal of Combinatorial Optimization, 15(1), pp. 17–48. Available at: https://doi.org/10.1007/S10878-007-9082-4.

Daka, E. and Fraser, G. (2014) “A Survey on Unit Testing Practices and Problems,” in 2014 IEEE 25th International Symposium on Software Reliability Engineering, pp. 201–211. Available at: https://doi.org/10.1109/ISSRE.2014.11.

Dan, H. and Hierons, R.M. (2011) “Conformance testing from message sequence charts,” Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation, ICST 2011, pp. 279–288. Available at: https://doi.org/10.1109/ICST.2011.29.

Dinella, E. et al. (2022) “TOGA: A Neural Method for Test Oracle Generation,” in 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), pp. 2130–2141. Available at: https://doi.org/10.1145/3510003.3510141.

Dumlu, E. et al. (2011) “Feedback driven adaptive combinatorial testing,” 2011 International Symposium on Software Testing and Analysis, ISSTA 2011 - Proceedings, pp. 243–253. Available at: https://doi.org/10.1145/2001420.2001450.

Dutertre, B. and de Moura, L. (2006) “A fast linear-arithmetic solver for DPLL(T),” Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 4144 LNCS, pp. 81–94. Available at: https://doi.org/10.1007/11817963_11.

Fard, A.M., Mesbah, A. and Wohlstadter, E. (2015) “Generating Fixtures for JavaScript Unit Testing (T),” in 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 190–200. Available at: https://doi.org/10.1109/ASE.2015.26.

Fraser, G. and Arcuri, A. (2011) “Evolutionary Generation of Whole Test Suites,” in 2011 11th International Conference on Quality Software, pp. 31–40. Available at: https://doi.org/10.1109/QSIC.2011.19.

Fraser, G. and Zeller, A. (2010) “Mutation-driven generation of unit tests and oracles,” ISSTA’10 - Proceedings of the 2010 International Symposium on Software Testing and Analysis, pp. 147–157. Available at: https://doi.org/10.1145/1831708.1831728.

Fraser, G. and Zeller, A. (2011) “Exploiting common object usage in test case generation,” Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation, ICST 2011, pp. 80–89. Available at: https://doi.org/10.1109/ICST.2011.53.

Gamoura, S.C. (2023) “Explainable AI (XAI) for AI-Acceptability: The Coming Age of Digital Management 5.0,” in ICNSC 2023 - 20th IEEE International Conference on Networking, Sensing and Control. Institute of Electrical and Electronics Engineers Inc. Available at: https://doi.org/10.1109/ICNSC58704.2023.10319030.

Ganesh, V. and Dill, D.L. (2007) “A decision procedure for bit-vectors and arrays,” Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 4590 LNCS, pp. 519–531. Available at: https://doi.org/10.1007/978-3-540-73368-3_52.

Ganov, S.R. et al. (2008) “Test generation for graphical user interfaces based on symbolic execution,” Proceedings - International Conference on Software Engineering, pp. 33–40. Available at: https://doi.org/10.1145/1370042.1370050.

Garvin, B.J., Cohen, M.B. and Dwyer, M.B. (2011) “Evaluating improvements to a meta-heuristic search for constrained interaction testing,” Empirical Software Engineering, 16(1), pp. 61–102. Available at: https://doi.org/10.1007/S10664-010-9135-7.

Gaudel, M.C. (1995) “Testing can be formal, too,” Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 915, pp. 82–96. Available at: https://doi.org/10.1007/3-540-59293-8_188.

Grano, G. et al. (2018) “An Empirical Investigation on the Readability of Manual and Generated Test Cases,” in 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC), pp. 348–3483.

Grunert, F.Paul. et al. (no date) Humanities and artificial intelligence.

Kim, S. et al. (2021) “Code Prediction by Feeding Trees to Transformers,” in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 150–162. Available at: https://doi.org/10.1109/ICSE43902.2021.00026.

Klima, M. et al. (2023) “Specialized path-based technique to test Internet of Things system functionality under limited network connectivity,” Internet of Things (Netherlands), 22. Available at: https://doi.org/10.1016/j.iot.2023.100706.

Lemieux, C. et al. (2023) “CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models,” in 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 919–931. Available at: https://doi.org/10.1109/ICSE48619.2023.00085.

Li, X. et al. (2024) “ESSENT: an arithmetic optimization algorithm with enhanced scatter search strategy for automated test case generation,” Information Sciences, 656. Available at: https://doi.org/10.1016/j.ins.2023.119915.

Ma, P. et al. (2020) “Can this fault be detected: A study on fault detection via automated test generation,” Journal of Systems and Software, 170. Available at: https://doi.org/10.1016/j.jss.2020.110769.

Maciej Serda et al. (2013) “Synteza i aktywność biologiczna nowych analogów tiosemikarbazonowych chelatorów żelaza,” Uniwersytet śląski. Edited by G. Balint et al., 7(1), pp. 343–354. Available at: https://doi.org/10.2/JQUERY.MIN.JS.

Marculescu, B. et al. (2016) “Tester interactivity makes a difference in search-based software testing: A controlled experiment,” Information and Software Technology, 78, pp. 66–82. Available at: https://doi.org/10.1016/j.infsof.2016.05.009.

Mastropaolo, A. et al. (2021) “Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks,” in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 336–347. Available at: https://doi.org/10.1109/ICSE43902.2021.00041.

Mastropaolo, A. et al. (2023) “Using Transfer Learning for Code-Related Tasks,” IEEE Transactions on Software Engineering, 49(4), pp. 1580–1598. Available at: https://doi.org/10.1109/TSE.2022.3183297.

McMinn, P. (2011) “Search-Based Software Testing: Past, Present and Future,” in 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, pp. 153–163. Available at: https://doi.org/10.1109/ICSTW.2011.100.

Mirshokraie, S., Mesbah, A. and Pattabiraman, K. (2013) “PYTHIA: Generating test cases with oracles for JavaScript applications,” in 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 610–615. Available at: https://doi.org/10.1109/ASE.2013.6693121.

Mirshokraie, S., Mesbah, A. and Pattabiraman, K. (2015) “JSEFT: Automated Javascript Unit Test Generation,” in 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), pp. 1–10. Available at: https://doi.org/10.1109/ICST.2015.7102595.

Palomba, F. et al. (2016) “On the Diffusion of Test Smells in Automatically Generated Test Code: An Empirical Study,” in 2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST), pp. 5–14. Available at: https://doi.org/10.1145/2897010.2897016.

Panichella, A. et al. (2020) “Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities,” in 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 523–533. Available at: https://doi.org/10.1109/ICSME46990.2020.00056.

Panichella, A., Kifetew, F.M. and Tonella, P. (2018) “Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets,” IEEE Transactions on Software Engineering, 44(2), pp. 122–158. Available at: https://doi.org/10.1109/TSE.2017.2663435.

Pezzè, M. and Zhang, C. (2014) “Automated test oracles: A survey,” Advances in Computers, 95, pp. 1–48. Available at: https://doi.org/10.1016/B978-0-12-800160-8.00001-2.

Prado Lima, J.A. and Vergilio, S.R. (2020) “Test Case Prioritization in Continuous Integration environments: A systematic mapping study,” Information and Software Technology, 121. Available at: https://doi.org/10.1016/j.infsof.2020.106268.

Ren, J. and Zhu, W. (2023) “Backtracking search optimization algorithm with dual scatter search strategy for automated test case generation,” Journal of King Saud University - Computer and Information Sciences, 35(7). Available at: https://doi.org/10.1016/j.jksuci.2023.101600.

Rezaalipour, M. and Furia, C.A. (2023) “An annotation-based approach for finding bugs in neural network programs,” Journal of Systems and Software, 201. Available at: https://doi.org/10.1016/j.jss.2023.111669.

Saeed, A., Ab Hamid, S.H. and Mustafa, M.B. (2016) “The experimental applications of search-based techniques for model-based testing: Taxonomy and systematic literature review,” Applied Soft Computing Journal, 49, pp. 1094–1117. Available at: https://doi.org/10.1016/j.asoc.2016.08.030.

Saxena, P. et al. (2010) “A Symbolic Execution Framework for JavaScript,” in 2010 IEEE Symposium on Security and Privacy, pp. 513–528. Available at: https://doi.org/10.1109/SP.2010.38.

Schafer, M. et al. (2023) “An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation,” IEEE Transactions on Software Engineering [Preprint]. Available at: https://doi.org/10.1109/TSE.2023.3334955.

Schuler, D. and Zeller, A. (2011) “Assessing Oracle Quality with Checked Coverage,” in 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, pp. 90–99. Available at: https://doi.org/10.1109/ICST.2011.32.

Shi, E. et al. (2023) “Towards Efficient Fine-tuning of Pre-trained Code Models: An Experimental Study and Beyond.” Available at: http://arxiv.org/abs/2304.05216.

Tufano, M. et al. (2022) “Generating Accurate Assert Statements for Unit Test Cases using Pretrained Transformers,” in 2022 IEEE/ACM International Conference on Automation of Software Test (AST), pp. 54–64. Available at: https://doi.org/10.1145/3524481.3527220.

Wang, J. et al. (2023) “Software Testing with Large Language Models: Survey, Landscape, and Vision.” Available at: http://arxiv.org/abs/2307.07221.

Weyssow, M. et al. (2023) “Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models.” Available at: http://arxiv.org/abs/2308.10462.

Wu, J. and Clause, J. (2023) “A uniqueness-based approach to provide descriptive JUnit test names,” Journal of Systems and Software, 205. Available at: https://doi.org/10.1016/j.jss.2023.111821.

Yu, H. et al. (2022) “Automated Assertion Generation via Information Retrieval and Its Integration with Deep learning,” in 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), pp. 163–174. Available at: https://doi.org/10.1145/3510003.3510149.

Zlotchevski, A. et al. (2022) “Exploring and evaluating personalized models for code generation,” in ESEC/FSE 2022 - Proceedings of the 30th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, Inc, pp. 1500–1508. Available at: https://doi.org/10.1145/3540250.3558959.treme Programming Explained: Embrace Change.. s.l.:Addison-Wesley.
Siddiqui, S., 2021. Learning Test-Driven Development. s.l.:O’Reilly.

J. Shore and S. Warden, 2021. The Art of Agile Development, Sebastopol. CA, USA: O'Reilly.
E. Daka and G. Fraser, "., 2014. A survey on unit testing practices and problems. s.l.:s.n.
M. M. Almasi, H. H. G. F. A. A. a. J. B., n.d. An industrial evaluation of unit test generation: Finding real faults in a financial application. s.l.:s.n.
Zalewski, M., 2023. American fuzzy lop. s.l.:s.n.
G. Grano, S. S. H. C. G. a. R. O., 2018. An empirical investigation on the readability of manual and generated test cases. s.l.:s.n.
A. Panichella, S. P. G. F. A. A. S. a. V. J. H., 2020. Revisiting test smells in automatically generated tests: Limitations pitfalls and opportunities. s.l.:s.n.
E. Daka, J. C. G. F. J. D. a. W. W., 2015. Modeling readability to improve unit tests. s.l.:s.n.
al., B. C. e., 2022. CodeT: Code Generation with Generated Tests. s.l.:s.n.
S. K. Lahiri et al., 2022. Interactive code generation via test-driven user-intent formalization. s.l.:s.n.
E. Arteca, S. H. M. P. a. F. T., 2022. "Nessie: Automatically testing JavaScript APIs with asynchronous callbacks", Proc.. s.l.:44th IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE), pp. 1494-1505,..
al., T. B. B. e., 2020. "Language models are few-shot learners". s.l.:Available: https://arxiv.org/abs/2005.14165..
Anon., n.d. "OpenAI LLMs: Deprecations", Nov. 2023, [online] Available: https://platform.openai.com/docs/deprecations.. s.l.:s.n.
"Starcoder: A state-of-the-art LLM for code" , Nov. 2023. s.l.:[online] Available: https://huggingface.co/blog/starcoder..
Anon., n.d. s.l.:s.n.
Saswat Anand, E. K. B. T. Y. C. J. C. M. B. C. W. G. M. H. M. J. H. P. M. A. B. J. J. L. H. Z., 2013. An orchestrated survey of methodologies for automated software test case generation, Journal of Systems and Software, Volume 86, Issue 8,.

APPENDIX A: RESEARCH PROPOSAL

APPENDIX B: PYTHON CODE FOR JSON PARSER

Please refer GitHub Link for details.

APPENDIX C: PYTHON CODE FOR MODEL IMPLEMENTATION

Please refer GitHub Link for details.

APPENDIX D: PYTHON CODE MODEL TESTING

Please refer GitHub Link for details