Assessing the Effectiveness of Fine-Tuned Large Language Models for Automated Unit Test Generation in Java: A Comparative Study

Final Project Report

Author: Reibjok Othow

Supervisor: Dr. Gunel Jahangirova

Student ID: 21009059

April 12, 2024

Abstract

Unit testing stands as a foundational practice in software development, serving as a crucial mechanism for validating the functional correctness and reliability of individual components in isolation from the broader system context. Despite its importance, manual unit testing often proves to be a labor-intensive process, posing challenges to the rapid progression of software development.

This report delves into the intersection of cutting-edge automation techniques and unit testing to address these issue, particularly focusing on the integration of Large Language Models (LLMs), exploring their potential to address existing challenges in unit testing. In particular, this study explores the efficacy of fine-tuned Large Language Models (LLMs) for automating unit test generation in Java, through a comprehensive comparative study. The study entails fine-tuning a large language model on open-source unit testing data extracted from five prominent GitHub repositories. The fine-tuned model is evaluated against Evosuite, a renowned search-based test generation tool in Java, an ordinary untrained LLM, and human-authored tests. These four test suites undergo analysis based on three critical criteria: coverage, mutation score, and adherence to a code convention.

The evaluation process involves prompting the model to generate tests for a Class Under Test (CUT), followed by iteratively re-prompting the model to rectify any errors arising from the generated tests. Upon error rectification, the findings indicate that the ordinary model outperforms the three other methodologies in terms of coverage. Conversely, Evosuite exhibits the highest mutation score, while the trained model surpasses other test suites in adherence to a code convention. Notably, the study reveals that the fine-tuned model does not exhibit superior performance over the ordinary model in any aspect.

This study sheds light on the strengths and limitations of fine-tuned LLMs in automated unit test generation, providing valuable insights for researchers and practitioners in the field of software testing and machine learning.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Reibjok Othow

April 12, 2024

Acknowledgements

I would like to thank my supervisor, Dr Gunel Jahangirova, who has provided me with invaluable help and support throughout the project. I could not have done this without her guidance and wisdom.

Contents

Chapter 1

Introduction

Unit testing is an essential stage of software development, serving as a keystone for validating the correctness and reliability of individual components within a program. This rigorous practice, while essential for software quality assurance, is not without its challenges, often emerging as a time-consuming process that can impede the fast progression of software development. As a result, the software development community has seen a surge in research and innovation directed at automating unit testing processes, with the aim of not only accelerating testing but also enhancing its efficiency and effectiveness.

While current automated testing frameworks have made strides in streamlining the generation and execution of test cases, they have inherent limitations which have prompted a search for more advanced solutions. One key limitation is that these automated test generation tools often require a reference implementation to compute oracles for the generated inputs [31]. Additionally, studies have highlighted issues with the quality of the test code generated by these tools, with evaluations showing the presence of suboptimal design choices, known as test smells, in the automatically generated unit test classes, such as in the case of Evosuite and Randoop which suffer Assert Roulette and Eager Test smells [20]. In addition, research has also shown that while tools like Randoop and Evosuite are used for automated unit test generation, there are challenges in how well the generated test suites perform at detecting actual faults in software systems [48]. The influence of automatically generated unit tests on software maintenance has also been another topic of investigation, indicating that the choice of traditional automation tools can impact the effectiveness of software maintenance activities [49].

This report delves into the convergence of the latest automation techniques, specifically the application of Large Language Models (LLMs) within the domain of unit testing. Large Lan-

guage Models, such as gpt-3.5-turbo, offer a promising avenue for overcoming these challenges, leveraging their language understanding capabilities to produce unit tests that are not only syntactically correct but also human-readable and contextually meaningful.

The landscape of LLMs for test generation encompasses many alternatives such as gpt-3.5-turbo [34], Codex [10], CodeGen [32] and BART [26] to name a few. These models exhibit diverse strengths based on their training data, architecture, and fine-tuning objectives, presenting varied options for developers seeking to automate unit testing in different contexts. As we shall explore in coming chapters, the effectiveness of LLMs in unit test generation has been a subject of scholarly exploration, revealing both their potential and limitations. By examining real-world results from empirical evaluations and comparative studies, this report draws on evidence to underscore the potential benefits and limitations associated with LLMs, setting the stage for the core focus of this work —— Assessing the Effectiveness of Fine-Tuned Large Language Models for Automated Unit Testing in Java: A Comparative Study. The strategic choice of Java, renowned for its clarity and robustness, aligns with the objective of generating high-quality, human-readable unit tests that adhere closely to coding conventions.

This report aims to unfold the rise of automated testing methodologies, the emergence of LLMs as a transformative force, and seeks to evaluate the combination the two for the task of unit test generation. The reader will be able to appreciate the rationale behind fine-tuning a Large Language Model to address the specific demands of unit test generation, exploring its advantages and limitations. Through this empirical analysis, the report contributes to the ongoing discourse surrounding automated testing methodologies, offering insights that pave the way for more efficient, readable, and contextually aware unit tests in the realm of software development.

1.1 Aims and Objectives

This project aims to evaluate a fine-tuned gpt-3.5-turbo model for automated unit test generation in the Java Programming Language. The overarching objectives of this report include:

- **1. Fine-Tune gpt-3.5-turbo for Unit Test Generation:** Recognising the need for tailored solutions, delve into the process of fine-tuning gpt-3.5-turbo specifically for unit test generation in Java. Utilise a curated dataset of high-quality test cases to optimise the model's capabilities, addressing challenges identified in existing literature.

- **2. Collect and Examine Empirical Results:** Gather and assess the outcomes of fine-

tuning gpt-3.5-turbo for unit test generation, considering factors such as coverage, quality, readability, correctness and adherence to coding conventions. Draw comparisons with an untrained gpt=3.5-turbo LLM and traditional automated testing methods, using Evosuite test generation tool [13] as a representative, to provide a comprehensive understanding of the model's performance.

Through the fulfilment of these aims and objectives, this project seeks to contribute valuable insights to automated testing techniques, paving the way for more efficient and readable tests.

Chapter 2

Background

This section will start by giving an overview of software testing and unit testing in particular, and discuss the tedious nature of manual testing. It will then continue with an exploration of the different methodologies of unit test automation that exist today and the advantages and short comings of each. Upon this, it will explore Large Language Models and the recent surge in their use for various tasks. In particular it will assess their ability to generate test cases and a review of recent work that have traversed their use for this task. Finally, it will make a case for fine tuning a Large Language Model for Test Generation, which is the heart of this project.

2.1 Software Testing

Software testing stands as a cornerstone within the realm of software engineering, representing a critical phase in the software development life cycle (SDLC). It embodies a multifaceted process essential for ensuring the reliability, functionality, and overall quality of software systems. At its core, software testing entails a methodical and systematic examination of software components and functionalities to validate their compliance with specified requirements and intended behaviour. This process is indispensable for detecting and rectifying defects, errors, and anomalies that may compromise the performance of the software system. By subjecting the software to an assortment of tests, ranging from unit-level inspection to comprehensive system-level evaluations, software testers aim to uncover dormant issues and mitigate potential risks before the deployment of the software system. Moreover, software testing serves as a channel for verifying adherence to design specifications, regulatory standards, and user expectations, thereby instilling confidence in the software's efficacy and suitability for its intended purpose.

2.2 Unit Testing

Unit testing, as a subset of software testing, occupies a pivotal position within this broader landscape, focusing on the granular examination of individual units or components within the software code-base. These units typically manifest as discrete classes, methods, or modules, each representing a cohesive and self-contained unit of functionality. Unlike other forms of testing that assess the integrated behaviour of the software system as a whole, unit testing isolates these constituent units for meticulous examination. Through the creation of targeted test cases and the execution of controlled tests, developers endeavour to validate the correctness, robustness, and reliability of each unit in isolation. By uncovering defects at the micro-level and addressing them early in the development cycle, unit testing contributes significantly to the overall enhancement of software quality and the alleviation of technical debt. This proactive approach not only fosters a culture of quality assurance but also promotes the long-term maintainability, scalability, and sustainability of software projects. However, the endeavour of unit testing is not without its challenges, as the painstaking crafting and execution of test cases can require considerable time and effort. Developers must navigate the delicate balance between thoroughness and efficiency, striving to achieve adequate test coverage without unduly impeding the pace of development. Nonetheless, the benefits accrued from diligent unit testing, in terms of improved software reliability, reduced defect density, and enhanced developer productivity, underscore its indispensable role within the software engineering paradigm.

2.3 Unit Test Quality Metrics

Several metrics and measurements are employed to gauge the adequacy, effectiveness, and comprehensiveness of unit testing efforts.

2.3.1 Code Coverage

One commonly used metric is code coverage, which quantifies the extent to which the source code of a software system is exercised by the unit tests. Code coverage metrics, such as statement coverage, branch coverage, and path coverage, provide insights into the proportion of code paths and logic branches that are traversed during testing.

1. **Statement Coverage:** Statement coverage measures the proportion of executable statements within the source code that are executed during the execution of the unit tests. It tracks whether each individual line of code has been executed at least once during testing.

Statement coverage is a basic metric that indicates the overall reach of the test suite in terms of executing different code paths and identifying potential defects. However, it does not account for all possible control flow paths within the code.

2. Branch Coverage: Branch coverage extends the concept of statement coverage by considering the execution of conditional branches and decision points within the code. It measures the percentage of decision outcomes or branches that are exercised by the unit tests. Branch coverage provides a more comprehensive assessment of test coverage compared to statement coverage, as it evaluates the testing of different logical conditions and control flow paths within the code. Achieving high branch coverage indicates that the unit tests are effective in evaluating the various decision points and logic branches present in the code base.

3. Line Coverage: Line coverage is similar to statement coverage but focuses specifically on the percentage of lines of code that are executed during testing. It considers each line of code, including comments and white space, and determines whether it has been executed by the unit tests. While line coverage provides a detailed view of code execution at the line level, it may not necessarily capture all control flow paths or decision points within the code base. Nonetheless, it serves as a useful metric for assessing the overall completeness and thoroughness of the test suite in terms of code execution.

While high code coverage does not guarantee the absence of defects, it serves as a useful indicator of the thoroughness of test coverage and identifies areas of the code base that require additional testing scrutiny.

2.3.2 Mutation Score

Another important metric is mutation score, which assesses the effectiveness of unit tests in detecting changes, or mutations, made to the source code. Mutation testing involves introducing small modifications, or mutations, to the code base and then running the unit tests to determine if they can detect these alterations. The mutation score reflects the percentage of mutations that are successfully detected by the test suite. A high mutation score indicates robust test coverage and suggests that the unit tests are adept at identifying potential defects introduced through code changes.

2.3.3 Fault Detection Rate

Furthermore, fault detection rate is a metric used to measure the efficacy of unit tests in uncovering faults or defects within the software. It quantifies the proportion of known defects

that are detected by the unit tests during testing. By comparing the number of defects identified by unit tests against the total number of defects present in the code base, developers can assess the fault detection capability of the test suite and prioritize efforts to improve test coverage in areas with higher defect density.

2.3.4 Test Suite Maintainability Metrics

Additionally, test suite maintainability metrics, such as test readability, test modularity, and test fragility, provide insights into the ease of maintaining and evolving the unit tests over time. Well-structured and maintainable unit tests are essential for facilitating ongoing development activities, promoting code refactoring, and accommodating changes to the software requirements or design.

2.4 Unit Test Automation

As the complexity of software systems and the meticulous nature of manual unit testing has increased over time, the need for efficient and scalable testing solutions has become paramount. Extensive research has been put into automatic test generation using techniques such as fuzzing [17] [39], mutative search-based techniques [13] [28], feedback-directed random test generation [41] [40], dynamic symbolic execution [18] [8] and hybrid techniques. In this section, we delve into the evolution of unit test automation tools, with a particular focus on Java, taking a particular interest in search based automation techniques.

2.4.1 Early Approaches to Unit Testing

The concept of unit testing dates back several decades, with early pioneers such as Kent Beck and Ward Cunningham advocating for the practice in the 1980s and 1990s. Initially, unit tests were written manually by developers, often as part of informal testing procedures. However, as software projects grew in size and complexity, manual testing became impractical, necessitating the automation of unit testing processes.

2.4.2 Introduction of JUnit

In the late 1990s, JUnit emerged as a groundbreaking framework for automating unit tests in Java. Developed by Kent Beck and Erich Gamma, JUnit introduced a simple yet powerful framework for writing and executing unit tests within the Java environment. By providing a

standardised approach to test organisation, assertion management, and test execution, JUnit revolutionised the way Java developers approached unit testing. Its influence extended far beyond the Java ecosystem, inspiring the creation of similar frameworks in other programming languages such as CppUnit for C++, PyUnit for Python and Nunit for .NET.

2.4.3 Advancements in Test Automation Tools

Following the success of JUnit, the landscape of unit test automation continued to evolve rapidly. Within the domain of Java, a multitude of automated unit testing tools have been developed, each aimed at refining the testing process and enhancing software quality such as Evosuite, Randoop, T3, JCrasher and AgitarOne to name a few. Among these tools, Randoop stood out as one of the early pioneers, employing techniques to delve into Java program behaviour through feedback-directed random testing. This methodology dynamically analyses feedback gathered from prior test runs to steer the generation of subsequent test cases, ensuring a thorough exploration of the program's functionality. In a similar vein, T3 (Test Template Toolkit) adopts a template-based synthesis approach to automatically create unit tests for Java classes. By allowing users to either specify or infer properties of the code, T3 facilitates for the systematic generation of tests tailored to the intricacies of the codebase. Contrasting T3, JCrasher ventures into test case generation by traversing the input domain of the program under test (PUT) utilising random and semi-random strategies. This process yields test suites that encompass diverse scenarios, encompassing edge cases and error conditions crucial for comprehensive testing. Concurrently, commercial offerings such as AgitarOne, previously known as Agitator, provide sophisticated automated unit testing capabilities for Java. This diverse array of automated testing tools collectively contributes to the refinement of software testing practices, in the next section we will taking a closer look at search-based approaches, a precursor to the discussion on advanced solutions like EvoSuite, which marks the segue into our exploration of Large Language Models for unit testing.

2.4.4 The Rise of Evolutionary Testing Approaches

Search Based Software Engineering (SBSE) techniques [21] have been employed in a vast array of tasks such as requirements optimisation, project planning and maintenance [33] [12] [58]. One area that they have proved to be particularly useful is software testing. Search-Based Software Testing (SBST) harnesses the power of evolutionary search algorithms to enhance the effectiveness of test case generation and optimisation [55]. It's approach defines software testing

as a optimisation problem [42] [43]. At its core, SBST employs genetic algorithms to iteratively evolve and mutate test cases over successive generations, aiming to maximise code coverage and defect detection. Various studies in the past have shown that SBST is more effective for identifying code defects than more traditional test generation techniques [15] [59]. Many SBST tools exist in Java [40] [11] [3] with the most popular and effective being EvoSuite [13] [5], which we will be using for our evaluation.

EvoSuite stands out as a pioneering approach to automated test generation, particularly for Java applications. Developed by researchers at the University of Luxembourg, EvoSuite employs these SBST approaches to automatically generate unit tests that achieve high code coverage and detect potential faults in the software [16], taking a fundamentally different approach to other previous tools like Randoop. By harnessing the power of genetic algorithms and search-based techniques, EvoSuite addresses the limitations of traditional test generation methods and offers a scalable and efficient approach to unit test automation, achieving higher coverage and mutation scores than previous approaches.

EvoSuite revolutionised the automated test generation process, offering a scalable and efficient approach to unit test automation. Its ability to generate high-quality test cases quickly and effectively has made EvoSuite a valuable tool for Java developers seeking to enhance the reliability and robustness of their software applications.

However, while current automated testing frameworks such as Evosuite have made strides in streamlining the generation and execution of test cases, they have inherent limitations which have prompted a search for more advanced solutions. One key limitation is that these automated test generation tools often require a reference implementation to compute oracles for the generated inputs [31]. Additionally, studies have highlighted issues with the quality of the test code generated by these tools, with evaluations showing the presence of suboptimal design choices, known as test smells, in the automatically generated unit test classes, such as in the case of Evosuite and Randoop which suffer Assert Roulette and Eager Test smells [20]. In addition, research has also shown that while tools like Evosuite are used for automated unit test generation, there are challenges in how well the generated test suites perform at detecting actual faults in software systems [48]. The influence of automatically generated unit tests on software maintenance has also been another topic of investigation, indicating that the choice of traditional automation tools can impact the effectiveness of software maintenance activities [49].

2.5 Large Language Models

In light of the identified limitations, there is a growing interest in employing deep learning based code generation techniques to enhance the creation of more effective unit tests, particularly using transformer-based models. Large Language Models (LLMs) have witnessed a recent surge in popularity, showcasing their proficiency across diverse tasks including but not limited to text generation, text summarisation, image generation, and code generation [7] [29] [30]. Large Language Models are characterised by their extensive neural architectures, typically composed of millions or even billions of parameters, enabling them to capture long-range dependencies and generalise well to different tasks.

Large Language Models (LLMs) represent a significant advancement in artificial intelligence and natural language processing. These models, such as OpenAI's GPT (Generative Pre-trained Transformer) series, are trained on vast amounts of text data and employ deep learning techniques to understand and generate human-like text. At the core of LLMs lie transformer architectures, which enable them to process and generate text by attending to contextual information across long sequences of words. This process involves multiple layers of self-attention mechanisms, where the model learns to assign different weights to input tokens based on their relevance to each other within the context of the sequence. Through extensive pre-training on diverse text corpora, LLMs acquire a broad understanding of language semantics, syntax, and pragmatics, enabling them to generate coherent and contextually relevant text across a wide range of topics and domains.

The versatility and sophistication of LLMs have led to various applications across different fields, including natural language understanding, text generation, machine translation, and more. In the context of software engineering, LLMs offer promising opportunities for automating various tasks, including code generation, code completion, code summarization, and even code testing. By leveraging their language understanding capabilities, LLMs can analyse and comprehend software artifacts, such as code snippets, documentation, and requirements specifications, to assist developers in their coding and testing efforts.

Several large language models have garnered attention for their potential applications in test generation. The now discontinued Codex [7], for instance, is renowned for its programming proficiency, as it was fine-tuned on a massive codebase. This specialisation makes Codex particularly adept at understanding and generating code-related content. On the other hand, models like BART (Bidirectional and Auto-Regressive Transformers) [18] excel in tasks involving sequence-to-sequence learning, making them valuable for tasks such as text summarisation

and paraphrasing. BART's specific architecture and pre-training objectives contribute to its effectiveness in these particular applications.

When comparing these alternatives with gpt-3.5-turbo, differences emerge in terms of training data, architecture, and fine-tuning objectives. gpt-3.5-turbo, as a variant of the gpt series, has been trained on diverse datasets covering a wide range of tasks, providing it with a broad understanding of language. Its architecture allows it to generate contextually relevant and coherent text across various domains.

2.5.1 Fine Tuning

Fine-tuning a large language model involves retraining the model on a specific dataset or task to adapt its parameters and representations for specialised domains or applications. This process typically involves initialising the pre-trained model with weights learned from a large corpus of general text data and fine-tuning it on a smaller, task-specific dataset through gradient-based optimisation techniques. Fine-tuning a large language model on domain-specific data allows the tailoring of its capabilities to better suit the nuances and intricacies of software-related tasks.

Chapter 3

Related Works

In laying the groundwork for a comprehensive understanding of this project's scope and objectives, it is imperative to embark on an exploration of the prior research that has paved the way for the integration of Large Language Models (LLMs) in the domain of automated unit testing. This can provide a crucial backdrop, offering insights into the evolution of automated testing methodologies and the specific challenges that have propelled the exploration of LLMs in this context. This section aims to do just this through an analysis of empirical evaluations, comparative studies, and theoretical published works. But first, we will look deeper into the workings of Evosuite, and look at a large scale study of the tool's effectiveness.

3.1 An Evaluation of Evosuite

Evosuite is an automated unit test generation tool designed for Java programs. It utilizes evolutionary computation techniques, specifically genetic algorithms, to automatically generate unit tests. First introduced by Gordon Fraser and Andrea Arcuri in the paper *Fraser et al. 2011* [13], the overarching goal of EvoSuite is to create comprehensive test suites that achieve high code coverage and effectively detect potential faults in the software under test.

The process of test generation in EvoSuite unfolds through a series of key stages. Initially, EvoSuite initializes a population of test suites, each containing a collection of test cases. These initial test suites can either be randomly generated or seeded with preliminary knowledge about the program under test. Following initialization, each test suite in the population undergoes fitness evaluation, where its effectiveness is assessed based on code coverage metrics such as branch or statement coverage. The overarching goal of EvoSuite is to maximize coverage while

minimizing redundancy in the generated tests.

Subsequently, test suites with higher fitness scores are selected to advance to the next generation, employing selection mechanisms inspired by principles of natural selection, such as tournament or roulette wheel selection[27][30]. Selected test suites then undergo genetic operations including crossover and mutation to produce offspring for the subsequent generation. Crossover involves combining elements of parent test suites to create new test cases, while mutation introduces random changes to explore new areas of the search space. These offspring test suites replace less fit individuals in the population, ensuring that only the fittest individuals propagate to subsequent generations.

The evolutionary process continues for a predefined number of generations or until a termination condition is met, such as reaching a target coverage threshold or exhausting computational resources. Throughout this iterative process, EvoSuite dynamically adapts and refines test suites based on feedback obtained from evaluating their fitness against the code under test. By systematically exploring the space of possible test cases, EvoSuite generates effective unit tests that provide thorough coverage of the program's functionality. This automation facilitates the swift creation of high-quality unit tests for Java programs, ultimately enhancing software reliability and reducing the likelihood of defects in production code

The research paper *Fraser et al. 2014* [16] presents an extensive evaluation of EvoSuite's automated unit test generation capabilities, focusing on branch coverage metrics and environmental dependencies across a diverse set of software systems. Utilizing the SF110 corpus, consisting of 110 open source projects, the study conducts empirical assessments to analyze EvoSuite's effectiveness in achieving branch coverage. The experiment encompasses various software artifacts, including open source, industrial, and automatically generated projects. EvoSuite demonstrates promising results, achieving an average branch coverage of 71% across the entire SF110 corpus. However, detailed analysis reveals variations in coverage levels based on the nature of software artifacts. For instance, classes without unsafe operations exhibit higher coverage rates, averaging at 84%, while classes involving network sockets demonstrate lower coverage, as low as 51%. This highlights the impact of environmental interactions on test outcomes and underscores the need for systematic artifact selection in empirical studies to ensure meaningful evaluations.

Methodologically, the study employs EvoSuite on the SF110 corpus and additional industrial systems to evaluate its performance. The experiment rigorously assesses branch coverage achieved by EvoSuite, providing quantitative insights into its effectiveness as an automated

testing tool. By systematically examining the practical implications of software artifact selection, the study elucidates the complexities and limitations associated with experimental design in empirical research. Moreover, the paper discusses challenges related to evaluating software artifacts and emphasizes the importance of clear and systematic selection criteria. Through numerical metrics and empirical analysis, the research contributes valuable insights into EvoSuite's capabilities in generating unit tests and highlights the critical role of environmental dependencies in test outcomes. These findings underscore the significance of thoughtful artifact selection in empirical studies, aiming to minimize biases and enhance the validity of research outcomes in software engineering.

3.2 Evaluations on Large Language Models

Some preliminary efforts have been made to utilize LLMs in the generation of unit tests. The study *Siddiq et al. 2023* [50] provides a comprehensive examination of the capabilities of large language models (LLMs) in generating unit tests, evaluating three prominent LLMs: ChatGPT-3.5 [34], Codex [10], and CodeGen [32]. Through systematic experimentation on datasets from the Multilingual HumanEval [7] and Evosuite SF110 [14] benchmarks, the study rigorously examines various facets of LLM-generated unit tests, including compilation rates, test correctness, coverage metrics, and the prevalence of test smells. Employing a structured approach, the research conducts data collection, unit test generation, and an evaluation, illustrating both the potential and limitations of LLMs for this task. They focused on two primary research questions - the proficiency of LLMs in generating unit tests and the influence of the inclusion of various code elements in the context on LLM performance - the findings underscore the complexities inherent in natural language understanding and code synthesis tasks, providing valuable insights for future research endeavours in AI-driven software engineering tools.

In their study, the authors meticulously collected data from the HumanEval and SF110 datasets, applying stringent criteria to select testable methods. For the SF110 dataset in particular, they applied inclusion and exclusion criteria for the selection of testable methods by filtering classes based on visibility and method characteristics, obtaining a subset of methods under test (MUTs) from a diverse range of projects. However, the collected data, comprising only 411 MUTs from 194 classes across 47 projects, reflects a relatively small-scale assessment within the expansive SF110 dataset (which contains 23,886 classes, with over 800,000 byte-code level branches and 6.6 million lines of code). In contrast, this report aims for a larger-scale evaluation, aiming to generate a test class for each testable class within a project in the SF110

dataset, similar to how Evosuite does it. Moreover, while the authors did not prompt the LLMs to fix compilation errors directly, resorting instead to heuristic fixes, my study seeks to address compilation errors by re-prompting the LLMs, thereby enhancing the robustness of the generated unit tests. Additionally, while the authors evaluated LLMs using a multilingual dataset encompassing problems in various programming languages, this study focuses solely on Java, thereby providing a more targeted evaluation within a specific programming context. These differences underscore distinct methodological approaches and research objectives, highlighting the unique contributions and considerations of each study.

None the less, this paper's study findings shed light on the performance of LLMs in generating unit tests across multiple dimensions. Notably, Codex, which is trained on a large codebase, emerged as the best performing model in terms of compilation rates. Before heuristics, Codex exhibited the highest compilation rates across both the HumanEval and SF100 datasets, surpassing other LLMs by a significant margin. Furthermore, Codex demonstrated superior correctness rates compared to ChatGPT and CodeGen, indicating its efficacy in producing more accurate unit tests. While the three LLMs are inherently different, these results indicate that further training a LLM can improve its performance, given that Codex was trained on a large codebase unlike the other two LLMs. However, despite Codex's performance advantages, all LLMs struggled to achieve high coverage metrics, with line and branch coverage rates considerably lower than manually authored tests and those generated by Evosuite. Additionally, the prevalence of test smells, such as Assertion Roulette and Empty Tests, underscored the challenges associated with ensuring the quality of LLM-generated unit tests. These findings highlight both the potential and limitations of leveraging LLMs for unit test generation tasks and provide valuable insights for future research directions in AI-driven software testing methodologies.

Another paper *Schafer et al. 2023* [47] conducted a similar evaluation, on gpt-3.5-turbo[34], code-cushman-002[35] and StarCoder[22] in generating Javascript test cases without requiring any additional training. Their approach involved providing the LLMs with prompts containing the signature and implementation of the function under test, along with usage examples extracted from documentation, and, in case of test failure, generating a new test by re-prompting the model with the failing test and error message. Comparing their results with Nessie [6], a feedback-directed Javascript test generation technique, the study evaluated its performance on 25 npm packages with a total of 1,684 API functions.

Overall, their results showed that the LLM-based testing techniques outperformed Nessie,

with the generated tests achieve significant coverage metrics, with a median statement coverage of 70.2% and branch coverage of 52.8%. In comparison, the state-of-the-art feedback-directed JavaScript test generation technique, Nessie, lags behind with only 51.3% statement coverage and 25.6% branch coverage. Notably, their evaluation also indicated that gpt-3.5-turbo outperformed the other LLMs with StarCoder (the smallest LLM) performing worst out of the bunch which in their words suggests that the effectiveness of the approach is influenced by the size and training set of the LLM, but does not fundamentally depend on the specific model. Among their evaluations was the assessment of the characteristic properties of the failing tests. Their results concluded that of a large amount of the failing tests were caused by correctness errors such as type, assertion, syntax and reference errors, as well as incorrect invocations and infinite recursions, a similar occurrence to *Siddiq et al. 2023* [50].

The study conducted by *Yu et al. 2023* [57] further delves into the utilisation of Large Language Models (LLMs), specifically employing ChatGPT as a representative, to address challenges within mobile application test script generation and migration. Their research aims to comprehensively investigate the capabilities and limitations of LLMs in this context. They observed that LLMs demonstrate proficiency in grasping the business logic of the Application Under Test (AUT) and dynamically adjusting the generation process based on the current state of the AUT. However, they also identified significant limitations, including issues related to context memory, API usage randomness, and the substantial human effort required to refine generated test cases. Their findings underscore the nuanced landscape of LLM-driven test automation, highlighting both the potential benefits and practical challenges associated with its application in the realm of mobile application testing and migration.

In contrast to the *Yu et al. 2023* [57] and *Schafer et al. 2023* [47] studies, my research focuses on a distinct domain within software testing, specifically Java unit testing, leveraging LLM technology. While all three studies share the common thread of exploring the capabilities of LLMs, my work diverges in terms of the application domain, methodology, and objectives. Whereas their study concentrates on mobile application testing and migration and within the domain of Javascript testing, this study is centered on Java unit testing, offering a novel perspective within the broader spectrum of LLM-driven software testing research.

The findings from the three evaluations explored in this section collectively offer valuable insights that can inform and enhance the methodology of this report in Java unit testing. Firstly, the discovery that LLMs, particularly specialized ones like Codex or larger models like ChatGPT, can effectively capture the business logic of a System Under Test (SUT) underscores

the potential of LLMs in generating meaningful and context-aware test cases. This finding suggests that integrating LLMs into the test generation process can potentially overcome the limi tations of other approaches like Evosuite, which may lack the ability to comprehend complex system behaviors comprehensively. Additionally, the observation that the performance of LLMs in test generation tasks varies based on factors such as size and specialization makes a case for tailoring an LLM to the specific testing domain, as this may yield better results. Hence, this study will the aim to leverage these insights to strategically select a LLM that is equipped with larger parameters and specialise it for Java unit testing tasks. However, before we can make a case for fine tuning, we must first explore prior studies that have evaluated LLMs against Search Based Software Testing (SBST) techniques, which this report aims to do. We will do this in the next section.

3.3 LLM vs SBST

Several papers have evaluated the effectiveness of Large Language Models against Search Based Software Testing methodologies for Unit Testing purposes.

In the study conducted by *Yutian et al. 2023* [53], a systematic comparison between test suites generated by OpenAI's ChatGPT LLM[34] and the SBST tool EvoSuite[13] was undertaken. The research aimed to evaluate critical factors such as correctness, readability, code coverage, and bug detection capability of the generated test suites.

In terms of methodology, the study by Yutian et al. (2023) involved a comprehensive evaluation process across a diverse set of Java programs. They conducted their evaluation on the DynaMOSA benchmark containing 346 Java classes sampled from 4 different datasets spanning 117 Java projects. Their evaluation methodology included assessing whether ChatGPT-generated test cases successfully compiled and executed, as well as analysing bug priority levels and patterns within the generated test suites. Additionally, the study utilized JaCoCo to measure instruction and branch coverage, allowing for a detailed comparison between the code coverage achieved by ChatGPT and EvoSuite.

Notably, their findings revealed that EvoSuite outperformed ChatGPT in terms of code coverage by 19%. To assess the adherence of ChatGPT-generated test cases to coding style conventions, the authors employed the state-of-the-art software quality tool CheckStyle. Their evaluation results indicated that ChatGPT does not have a specific code style that it consistently follows when generating test cases.

In their study, the researchers crafted a prompt by analysing phrases used in various tools

leveraging LLMs and verifying them with ChatGPT. This process involved collecting phrases from sources such as Google, Google Scholar, GitHub, and technical blogs, and then refining them. Ultimately, they combined three representative expressions into a single prompt: "Write a JUnit test case to cover methods in the following code (one test case for each method): $input?". Notably, the researchers clarified that their goal was not to create the best-performing prompt but rather to establish a reasonable one for ChatGPT usage in practical scenarios.

In contrast, beyond the divergence in fine-tuning a LLM, this report also intends to diverge in terms of its approach to prompt generation. While they derived their prompt from existing sources on the internet, this report aims to generate test cases using a default prompt and then refine it through prompt engineering. By learning from the generated tests, it seek to iteratively improve and optimise the prompt for better performance. This distinction underscores differing methodologies between the two studies.

In *Lemieux et al. 2023* [25], the researchers proposed a hybrid technique called CODAMOSA, which combines Large Language Models (LLMs), particularly OpenAI's Codex, with Search-Based Software Testing (SBST). The algorithm, evaluated over 486 benchmarks, aims to enhance coverage in automated test case generation. The approach monitors SBST's coverage progress and, upon stalling, utilises Codex to generate example test cases for undiscovered functions, thereby guiding SBST to more productive areas of the search space. The evaluation revealed that CODAMOSA achieved statistically and significantly higher coverage on numerous benchmarks compared to SBST and LLM-only baselines. Through a detailed investigation, Lemieux et al. 2023 addressed several research questions, including the comparison of CODAMOSA to baselines, the impact of design decisions on test effectiveness, qualitative analysis of coverage results, and the similarity of Codex-generated tests to out-of-prompt code. Notably, the tool was evaluated over 486 benchmarks and showed that CODAMOSA outperformed the SBST and LLM-only baselines on a significant portion of benchmarks, indicating its effectiveness in improving coverage. Additionally, design decisions such as uninterpreted statements, Codex hyper-parameters, low-coverage targeting, and prompting strategies were found to influence test effectiveness, with sampling Codex at a higher temperature showing consistently positive effects. Furthermore, case studies highlighted the importance of special strings and backup callables in contributing to coverage improvement, while an assessment of Codex-generated tests revealed their dissimilarity to out-of-prompt code, suggesting CODAMOSA's robustness across diverse scenarios.

In the next section, we will take a look at studies that have evaluated fine tuned Large

Language Models for the task of unit test generation.

3.4 Evaluations on Fined Tuned Models

In the *Tufano et al. 2020* [56] paper, the authors present ATHENATEST, an automated approach for generating unit test cases by leveraging sequence-to-sequence learning with the BART transformer model [26]. The methodology involves a two-step training process: denoising pretraining on large unsupervised English and Java corpora, followed by supervised fine-tuning for the downstream task of generating unit tests on the METHODS2TEST data set [29]. Using this approach, ATHENATEST is capable of producing thousands of syntactically correct, compilable, and effective test cases for Defects4j projects, invoking various testing APIs. The generated test cases demonstrate comparable test coverage to EvoSuite and are preferred by professional developers due to their realism, accuracy in asserting expected behaviour, and human-readable code structure according to their study.

The experimental design encompasses several research questions aimed at assessing the performance and effectiveness of ATHENATEST. Through rigorous evaluations, the authors investigate the impact of model pretraining, focal context selection, and test case quality analysis. The results indicate that pretraining the model on both English and Java corpora significantly enhances its performance in generating unit test cases. Moreover, incorporating additional focal context beyond the focal method such as focal class signature, method signatures and package declarations leads to improvements in model training and test case generation quality. ATHENATEST demonstrates its capability to generate correct and effective test cases for Defects4j projects, outperforming alternative approaches like EvoSuite and GPT-3 in terms of test coverage and developer preference. Overall, ATHENATEST presents a promising solution, offering realistic, accurate, and human-readable test cases that align with developers' expectations and preferences.

Other papers have sought to fine tune LLMs for assertion generation in order to achieve better results. For example, *Alagarsamy et al. 2023* [4] motivated by ATHENATEST, proposed A3Test, which augmented by assertion knowledge with a mechanism to verify naming consistency and test signatures, leverages domain adaptation principles to adapt existing knowledge from assertion generation to the test case generation task. The paper presented results comparing A3Test with the baseline approach, ATHENATEST, using the Defects4j data set [24]. A3Test achieves 147

In contrast to ATHENATEST, which focuses on generating unit tests at the method level,

this report diverges by aiming to generate tests for entire focal classes. While ATHENATEST adopts a sequence-to-sequence transformer model, specifically the BART model, this study utilizes the more expansive gpt-3.5-turbo model. Notably, the gpt-3.5-turbo model is substantially larger and more powerful compared to BART, potentially offering enhanced capabilities. Furthermore, unlike ATHENATEST, which undergoes pretraining on English and Java corpora, this report foregoes such pretraining steps, as the gpt-3.5-turbo model does not necessitate this additional training. Consequently, by leveraging gpt-3.5-turbo and focusing on entire focal classes rather than individual methods, this study seeks to explore an alternative avenue for automated test case generation with the potential for broader applicability and effectiveness in software testing practices.

3.5 Wrap up

In concluding this comprehensive exploration of prior works, we delved into several evaluations and studies that employed LLMs in the realm of unit test generation. Despite initial promise, scrutiny of LLMs' performance in the investigated works underscores their tendency to produce syntactically and logically incorrect test cases, and often performing poorer that SBST approaches in terms of coverage. We also discussed the distinctions between the various studies and this report, highlighting the alternative avenues in which this report aims to explore. In short, none of the evaluations, aimed to generate a project level test suite using a fine tuned LLM that is prompted to generate tests for entire focal class at a time. The next chapter defines the requirements of this project in more detail.

Chapter 4

Requirements

This section lists the requirements that govern the development of the deliverables of this project. The successful completion of this project aims to evaluate a fine-tuned gpt-3.5-turbo model for the task of unit test generation in Java, comparing its tests against those generated by an ordinary gpt-3.5-turbo model, Evosuite as well as manually authored tests.

4.1 Functional Requirements

Functional requirements enumerate the functionalities that the system must implement. This includes functionalities that the model, parser, or user interface should be able to execute. These are as follows:

1. The generated tests should be able to compile without throwing any errors
2. The generated tests should be able to run
3. The generated tests should be achieve high coverage
2. The Model should be able to be prompted through API calls
3. The Model should be able to generate and return several test cases when prompted with a class
4. The Parser should be able to find additional contextual information in the source code harbouring the class. This includes class signatures, method signatures, class properties, package declarations etc.
5. The Reprompter shoulld be able to

4.2 Non-functional Requirements

Non-functional requirements are formulated to uphold the overall quality of the system:

1. Code Style - The generated tests should follow a concise coding style. In other words, they should be well documented, and neatly written. Classes and methods should have appropriate names.

2. Reliability - The system should be designed to handle and recover well from errors, providing informative error messages to users. The trained model should demonstrate stability, generating consistent and reliable test cases across different API calls.

3. Usability - The GUI should provide an intuitive and user-friendly experience, with clear instruction and guides on how to generate tests.

4. Maintainability - The system should be designed with modular components, facilitating updates and maintenance.

5. Ethical Considerations - Develop and implement guidelines for the ethical use of the trained model, including considerations related to bias and fairness in test case generation.

Chapter 5

Design & Specification

5.1 Overview

In this section we delve into the design specifications that need to be implemented, serving as a comprehensive roadmap that details the approaches necessary to fulfill the requirements outlined in the previous chapter effectively. The design of this project is divided into two stages, the fine tuning stage and the evaluation. Before delving into the detailed design of each stage, we must first establish a foundational understanding of the preliminary choices and prerequisites that inform our approach.

5.1.1 Programming Language

As previously stated, the unit tests will be generated in the Java programming language. The decision to generate unit tests in the Java is primarily motivated by strategic considerations. Firstly, Java has been extensively utilized in prior research endeavours within the domain of unit test generation. Moreover, it's notable that the majority of Search-Based Software Testing (SBST) techniques rely on the Java programming language. Additionally, Java is renowned for its robustness, static typing, and clear, rigorous syntax, offering distinct advantages for both the generation and evaluation of test cases. The language's rigorous structure requires generated tests to adhere closely to coding conventions, thereby enhancing the readability and understandability of the generated tests. Furthermore, Java's static typing characteristic provides a well-defined structure to the code, facilitating precise evaluation metrics and ensuring adherence to the language's stringent typing rules. These characteristics collectively position Java as an optimal choice for the unit test generation using a LLM, offering a controlled

environment for precise evaluation and fostering the generation of high-quality and syntactically accurate test cases.

5.1.2 Large Language Model

The selection of gpt-3.5-turbo as the underlying model is primarily driven by two significant factors. Firstly, gpt-3.5-turbo has been extensively utilized in prior research within the field. This extensive usage underscores its reliability and effectiveness in various natural language processing tasks, including unit test generation. Secondly, gpt-3.5-turbo enjoys widespread popularity worldwide, with over 180 million users, solidifying its position as an industry-leading Large Language Model (LLM). Additionally, the model possesses a formidable architecture, boasting an impressive 175 billion parameters, further enhances its capabilities. This expansive design not only signifies its state-of-the-art capabilities but also positions it as a formidable candidate for capturing intricate language and code nuances. The sheer scale of its parameters grants gpt-3.5-turbo an unprecedented ability to understand and generate contextually rich and coherent text across diverse domains, including the complex landscape of unit test scenarios.

5.2 Fine Tuning

The initial phase of this project entails the fine-tuning of the gpt-3.5-turbo model to address the specific requirements of the task at hand. The fine-tuning stage is structured into three distinct phases:

1. Data Collection
2. Data Preparation
3. Fine Tuning

5.2.1 Data Collection

The data collection stage involves the extraction of high-quality test classes and their corresponding focal classes. In this context, a focal class represents the entity or component within a software project that a test aims to assess or verify, and typically encapsulates a specific module or class within the codebase. The collected test and focal class pairs will be selected from open source Java repositories on GitHub, adhering strictly to an open-source framework and governed by the Apache License, Version 2.0[1] or the GNU General Public License[2].

To ensure the collection of high quality data, the selection process prioritises repositories with the highest degree of community validation, only considering those that are among the most starred on Github. Following these guidelines, five repositories were chosen, and an additional supplementary project containing implementations of data structures and algorithms in Java was augmented. These projects are listed below:

1. Rxjava [46] (47.6k stars) RXJava is a library for asynchronous and event-based programming using observable sequences/streams.

2. Retrofit [52] (42.6k stars) Retrofit is a type-safe HTTP client for Android and Java by Square, Inc.

3. Springboot [45] (72.5k stars) Spring Boot makes it easy to create Spring-powered, production-grade applications and services with absolute minimum fuss.

4. Spring-Framework [51] (54.9k stars) Spring provides everything you need beyond the Java language to create enterprise applications in a wide range of scenarios and architectures.

5. Google Guava [23] (49.3k stars) This is a set of core libraries from Google that include collections, caching, primitives support, concurrency libraries, and more

6. TheAlgorithms [54] (56.4 stars) An educational repository contained implementations of data structures and algorithms in Java

For each selected project, the data collection strategy revolves around parsing the source and test directories to identify test classes and their respective focal classes. Test and focal classes will be matched in accordance with Java naming conventions, where test classes are named by suffixing the name of the focal class with "Test". For instance, a focal class named "ExampleClass" would be paired with a test class named "ExampleClassTest". Given that it is also a contention that files are named after the classes that they contain, the same is true for file names too.

It is worth noting that while every test class must have a focal class, not all focal classes must have a corresponding test class. Therefore, the focus will be on identifying test classes and finding their corresponding focal classes.

5.2.2 Data Preparation

Before the model can be trained on the data, it needs to be processed into a suitable format. OpenAI's documentation [36] specifies that the data must be formatted in a conversational chat

style in JSONL format. Each example in the dataset should resemble a conversation, structured as a list of messages where each message has a role, content, and optional name.There are three roles, the system role represents information providing default or initial context to guide the conversation. The user role signifies messages sent by the user, typically prompting the model to perform a specific task or action. Conversely, the assistant role denotes responses generated by the model itself, providing solutions or answers to the user's prompts.

During this stage, we will structure each mapping in the collected data into a default format so that there are two messages: one representing the user, prompting the model to generate a test case for the focal method, and the other representing the assistant, which provides the correct test case. An example of this formatting is provided below:

```
{
"messages": [
    {"role": "system", "content": "You are a proficient Java developer"},
    {"role": "user", "content": "*PROMPT* <<FOCAL CLASS>>"},
    {"role": "assistant", "content": "<<TEST CLASS>>"}
]
}
```

Additionally, only collected pairs that are below the 16,345 token limit will be added to the JSONL file. To manage this, a token counter is implemented using the tiktoken library. This counter will also be utilized to calculate the total number of tokens in the training set and estimate the associated cost.

Once each mapping in the collected data has been structured into the default format and filtered based on token limit criteria, the entire formatted dataset will be compiled and organized into a JSONL (JSON Lines) file. This file will serve as the standardized format for the training data, suitable for subsequent fine-tuning stage.

5.2.3 Fine Tuning

Once the training set is prepared, the fine-tuning process of the model ensues following the guidelines outlined in the documentation. The fine-tuning process commences by uploading the formatted JSONL file to OpenAI's platform using the Files API. This step initiates the processing of the file and returns a training file ID. The file upload task can take several minutes, and while it undergoes processing, subsequent actions such as creating a fine-tuning job on the gpt-3.5-turbo model can be started. However, it's imperative to note that the fine-tuning job

will commence only after the file processing is complete. Upon starting the fine-tuning job, the model training process commences. For this project, the model will be trained over three epochs. After the fine-tuning is concluded, a model ID and a fine-tuning job report is produced and are available on the OpenAI platform. This report contains the training loss, trained tokens, and other relevant statistics that were a result of the fine tuning of the model.

5.3 Evaluation

5.3.1 Overview

The evaluation stage serves as the second phase of this report, following the fine-tuning process. In this stage, the performance of the fine-tuned model will be rigorously assessed on the widely recognised SF110 EvoSuite dataset against three benchmarking approaches:

1. Evosuite, a search-based test generation suite for Java.
2. An untrained gpt-3.5-turbo model, serving as a baseline comparison.
3. Manually authored tests, representing conventional testing practices.

Prompt engineering techniques will be employed to optimise the setup for generating test cases using the fine-tuned model. Furthermore, failing tests generated by both large language models will be reprompted to address any errors that were thrown during their compilation.

The evaluation will involve an inspection of the characteristics of any errors generated by the tests, providing insights into potential weaknesses or limitations of large language models. After the handling of these failing errors, the tests generated by all four approaches will undergo a comprehensive evaluation based on the following criteria:

1. Coverage
2. Test Correctness
3. Mutation Score
4. Readability

Before we proceed into the experimental design and setup, it is essential to delve into the specifics of the dataset that will be used for the evaluation. The following subsection will provide a comprehensive exploration of the SF110 dataset, laying the groundwork for the subsequent experimental procedures.

5.3.2 The Evosuite SF110 Benchmark

As previously mentioned, the fine tuned model will be evaluatd on the SF110 benchmark. This dataset, established by Evosuite, stands as a widely recognized benchmark for evaluating automated test generation techniques within the Java domain. Initially compiled and curated by *Fraser et al.(2012)* [14], the dataset originated from a collection of 100 Java projects sourced from SourceForge, a prominent open-source repository hosting over 300,000 projects and boasting more than two million registered users. This compilation was later augmented with an additional 10 more popular projects, resulting in the expanded corpus known as SF110. The dataset's inclusion of a myriad of Java projects, spanning from small-scale applications to large-scale software systems, renders it an invaluable resource within the research community. Due to this extensive coverage and popularity in previous research endeavours [16][50][53][25][56], the SF110 benchmark has become a cornerstone in evaluating the effectiveness and efficiency of various test generation approaches.

The SF110 benchmark dataset is known for its diversity, encompassing projects from different domains and exhibiting varying characteristics in terms of size, complexity, and functionality. This diversity makes it an ideal choice for evaluating the robustness and versatility of test generation techniques across a broad spectrum of Java applications.

However, due to resource and budget constraints, the evaluation in this study will focus on a subset of the SF110 benchmark, consisting of 32 projects carefully selected to maintain diversity while ensuring feasibility of evaluation. The selected subset includes both large and smaller-scale projects, ensuring representation across different project sizes and complexities.

The selected projects for this subset are as follows:

1. Tullibee	9. NekoMUD	17. openjms	25.
2. Rif	10. Saxpath	18. Ext4J	26.
3. TemplateIt	11. JNI-InChI	19. Io Project	27.
4. FalseLight	12. Xisemele	20. Wheel	28.
5. imSMART	13. PetSoar	21. JavAthena	29.
6. jdbacl	14. Follow	22. xBus	30.
7. Inspirento	15. Asphodel	23. T-Robots	31. sweethome3d
8. JSecurity	16. Lavalamp	24. HEAL	32. Weka

It's important to note that all projects in the SF110 benchmark dataset were massaged into a common Apache Ant build infrastructure to ensure consistency and reproducibility of the evaluation process. This common build infrastructure will be further elucidated in the subsequent subsection, providing insights into the preprocessing steps undertaken to prepare the dataset for evaluation.

5.3.3 Apache Ant

Apache Ant is a Java-based build tool that primarily focuses on automating software build processes. It serves as a popular choice for managing the build lifecycle of Java applications along with Maven and Gradle, providing developers with a flexible framework for compiling, testing, and deploying software projects. As an open-source tool maintained by the Apache Software Foundation, Apache Ant is widely adopted within the Java development community, and is the build tool of choice for the SF110 Dataset.

Apache Ant operates through the definition of build scripts, typically written in XML format, which specify the sequence of tasks to be executed during the build process. These tasks can include compiling source code, running tests, packaging binaries, and deploying artifacts to name a few. Each task is encapsulated within a target element, which defines its name and dependencies.

Understanding Apache Ant is pivotal for establishing a robust build infrastructure to handle projects from the SF110 dataset effectively. The familiarity with build automation principles provided by Apache Ant will facilitate the integration of these projects into the evaluation framework, enabling seamless test generation and execution processes.

5.3.4 Evosuite Test Generation Setup

Evosuite determines which classes are testable by analyzing the bytecode of Java classes within the Program Under Test. Classes that can be instantiated without causing exceptions are considered testable, enabling Evosuite to focus its efforts on generating tests for these classes. Within the evaluation dataset for this report, there are 3529 testable classes.

By default, Evosuite is configured with parameters and target criterion aimed at guiding its test generation. These parameters include settings for test generation strategies, coverage criteria, and search budgets. The default goals of Evosuite encompass objectives such as achieving high line and branch coverage, fault detection, and maximising diversity in generated test cases.

However, due to resource constraints, particularly limited computational resources, test generation with Evosuite needs to be constrained. To address this, a two-minute timeout per test case will be imposed during test generation. Given that the evaluation dataset contains a significant number of testable classes, optimising test generation efficiency is paramount.

Evosuite provides flexibility in how tests can be generated, offering options for command-line usage or integration with build tools like Maven. Since the dataset employs Apache Ant rather than Maven, tests will be generated using the command line for each project. This involves targeting each project's .jar file with Evosuite.

Utilizing Evosuite version 1.2.0, chosen for its compatibility with Java 11, the following command will be used to initiate test generation:

```
java -jar evosuite-1.2.0.jar -target <<project-jar>>.jar -Dsearch_budget=120
```

-Dsearch_budget sets the time out for each test case generation to 2 minutes. The remaining parameters, which are specified in an evosuite.properties file within each project, will be left to their default values.

Upon completion, a new directory named 'evosuite-tests' will be created within each project, containing the generated tests. Each testable class generates two files: a test class file and a scaffolding test class file, which are required to be able to execute the tests. Furthermore, an 'evosuite-report' directory is generated, housing a CSV file with statistics about the generated tests.

5.3.5 LLM Test Generation Setup

The generation of test cases using the Language Model (LLM) entails a structured process adhering to the requirements of OpenAI's API. Each project within the dataset comprises a 'src' directory housing two essential subdirectories: 'main/java', containing the primary source code, and 'test/java', housing human-authored tests designed for the primary source code. It is imperative to clarify that the focus of test generation in this study will be on files located within the 'main/java' path using the models and the tests located in 'test/java' will be used in the evaluation.

Test Directory Structure Creation

Initiating the test generation process involves the creation of a dedicated directory to store all generated tests. In accordance with Java conventions, test directories should mirror the structure of the classes they aim to test. For instance, a Java class named 'ExampleClass' within

the 'main/java/com/example' directory should have its corresponding test class in 'test/java/-com/example', named 'ExampleClassTest'.

For clarity, consider the following illustrative example:

```
main/java/
└── com
    └── example
        └── ExampleClass.java


trained_tests/
└── com
    └── example
        └── ExampleClassTest.java
```

Tests generated by the fine-tuned model will be stored in a directory named 'trained_tests', while those from the ordinary model will reside in 'untrained_tests'. The generation process entails parsing the source directory of each project, wherein for each '.java' file encountered, a corresponding '.java' file is created within the test directory. The newly generated test file mirrors the original file's directory structure but resides within the test directory, and is named after the original file but suffixed with 'Test' at the end of the name, as shown in the above example where "ExampleClassTest.java" is derived from "ExampleClass.java".

Prompting of the Models

As detailed in the data preparation subsection of the fine-tuning stage, OpenAI's API requires requests to be formatted in a conversational chat style using JSON format. Each request, structured as a conversation, comprises two messages: one from the system role providing contextual guidance, and the other from the user role containing the prompt. To conform to these specifications, each file within the source directory with a '.java' extension undergoes three steps. Firstly, the code within the file is extracted, followed by the removal of extraneous metadata comments (such as copyright declarations, authors etc.) to optimise token usage. Subsequently, this extracted code is appended to a JSON conversational template to be prompted to the LLM, as depicted below:

```
{
"messages": [
    {"role": "system", "content": "You are a proficient Java developer"},
    {"role": "user", "content": "Generate a Java test class using
    JUnit4 for the following Java class, reply with code only:
    **extracted code here**"}
]
}
```

While prompt engineering techniques will be applied iteratively, this initial prompt serves as the starting point. The JSON is then submitted via the OpenAI API, specifying the model (either gpt-3.5-turbo or the ID returned from fine-tuning). The API response, structured as JSON data, is processed to extract the response text, typically enclosed within "“‘" symbols. Upon extraction, this code is written to its corresponding test file. This process is repeated for all eligible Java files within the source directory, adhering to token constraints.

5.3.6 Error Analysis Setup

The Error Analysis Setup section delineates the meticulous process adopted to dissect errors emanating from the compilation of test cases. This multifaceted procedure encompasses three fundamental steps: Test Compilation and Logging, Error Extraction, and Statistics and Graph Generation.

1. Test Compilation and Logging

To scrutinize the errors engendered by the test cases, the compilation process is instigated through the Java compiler ('javac'). Within each project's 'build.xml' file, two Apache Ant tasks—namely 'compile-trained' and 'compile-untrained'—are appended. These tasks undertake the compilation of the trained tests stored in the 'trained_tests' directory and the untrained tests in the 'untrained_tests' directory, respectively.

The Apache Ant compilation tasks are executed programmatically leveraging Python's 'subprocess' library to capture of standard output (stdout) during task execution. This is leveraged in a ErrorLogger class which calls these tasks and captures the output produced during their execution. This logging mechanism allows for the recording of any errors encountered during the compilation process, which is pivotal for subsequent analysis. Below is the implementation

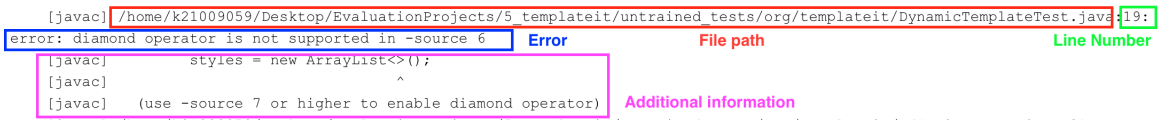
of the ErrorLogger class for the trained tests. The same implementation is replicated for the untrained tests.

```
class ErrorLogger:
    def log_compilation_errors(projects_directory):
        projects = os.listdir(projects_directory)

        for project in projects:
                        project_path = os.path.join(projects_directory, project)
            command_trained = ['ant', '-f', project_path, 'compile-trained']

            try:
                subprocess.run(command_trained, check=True,
                stdout=subprocess.PIPE, stderr=subprocess.PIPE,
                universal_newlines=True)
            except subprocess.CalledProcessError as e:
                            #If an error is thrown, log the standard output
                project_path = os.path.join(projects_directory, project)
                output_message = e.stdout if e.stdout is not None else ""
                log(output_message, project_path){e}")

    def log(output, project_path):
        file_name_trained = 'trained_compilation_errors.txt'
        file_path_trained = os.path.join(project_path, file_name_trained)

        with open(file_path_trained, 'w') as file_trained:
            file_trained.write(output)
```

2. Error Extraction

The subsequent step entails the extraction of errors from the logged standard output in the text files. Conventionally, when an Ant task is executed, it adheres to a sequence of dependencies. For ‘javac‘ compilation tasks, errors are presented in a specific format, encompassing attributes such as file path, line number, error type, and additional error

details. Below is an example of an error thrown in the terminal by the java compiler:



An 'ErrorFinder' class is instantiated to parse the logged text files and extract errors using a regex pattern shown below:

```
pattern = re.compile(r'\[javac\] ([^:\n]+):(\d+): error: (.+)')
```

This pattern matches the structure of the 'javac' errors as shown above, enabling the extraction of relevant error information. Subsequently, the extracted errors are encapsulated as objects of the 'Error' class, which encompasses properties such as file path, line number, error type, and error details. Additionally, the Error class provides methods to accommodate supplementary properties, facilitating a nuanced analysis.

```
class Error:
    def __init__(self, file, line, error, details):
        self.file = file
        self.line = line
        self.error = error
        self.details = details
        self.additional_properties = {}

    def set_additional_property(self, key, value):
        self.additional_properties[key] = value

    def get_additional_property(self, key):
        return self.additional_properties[key]

class ErrorFinder:
    def find_errors(directory_path):
        total_errors = []
        # Iterate through each directory in the specified directory
        for subdir in os.listdir(directory_path):
```

```
            subdir_path = os.path.join(directory_path, subdir)
            if os.path.isdir(subdir_path):
                error_file_path = os.path.join(subdir_path,
                'untrained_compilation_errors.txt')
                if os.path.exists(error_file_path):
                    with open(error_file_path, 'r') as file:
                        text = file.read()
                        errors = find_errors_through_regex(text)
                        total_errors += errors

    return total_errors
```

3. Statistics and Graph Generation

The final step in the error analysis process involves the generation of comprehensive statistics and illustrative graphs for in-depth analysis. The ‘ErrorFinder‘ class undertakes a comprehensive analysis of the extracted errors to compute statistics such as the total number of errors, the number of unique error types, the total number of failing test files as well as the count for each error type.

Furthermore, a ‘GraphGenerator‘ class is employed to utilise the extracted errors in generating graphical representations illustrating error distributions and trends. Prior to graph generation, certain error types undergo generalisation to ensure a more holistic analysis. These generalised errors serve as the foundation for crafting statistics and visual representations, enabling the identification of prevalent error patterns and trends within the dataset. An example is that an error such as ‘package Mockito does not exist’ is generalised to ‘package SomePackage does not exist’ to group all errors of this format.

This elaborate framework ensures a thorough and diligent analysis of errors encountered during test compilation, thereby furnishing invaluable insights for the evaluation process.

5.3.7 Prompt Generation Setup

The Prompt Engineering Setup describes the methodology devised to craft an optimised prompt for querying the Large Language Models to generate tests. The primary objective of this subsection is to establish a framework designed to generate an initial prompt that can yield test cases compiling without necessitating heuristic interventions.

The Prompt Engineering Setup is an essential component of the evaluation process, aiming to streamline the test generation process by calibrating the prompt used to query the LLMs. Initially, the approach entails generating an initial set of test cases, as described in the test generation setup, utilising the prompt

```
Generate a Java test class using JUnit4 for the following Java class,
reply with code only:
```

Subsequently, an error analysis is conducted as described in the previous subsection to determine prevalent errors encountered during test compilation.

Based on the errors identified during the test analysis phase, a systematic process of prompt refinement ensues, wherein contextual information is judiciously included into the prompt, to address errors resulting from not enough contextual information being provided in the previous prompt. The goal is not to find a prompt that eliminates all errors, as this is infeasible due to the impossibility of providing all the necessary context to eradicate errors entirely. Instead, the objective is to establish a good baseline that furnishes the LLMs with sufficient context to generate a test case for any Java class. The contextual information primarily revolves around test setup rather than the specifics of the test class itself.

5.3.8 Error Ratification Setup

The Error Ratification Setup seeks to explain the procedure devised to handle errors encountered during the compilation of test cases that were generated by the Large Language Models, incorporating a systematic approach to rectify compilation errors and ensure the generation of high-quality, compilable test suites

It constitutes a critical aspect of the evaluation process, aimed at mitigating errors encountered during the compilation of test cases generated by Large Language Models (LLMs). By reprompting the Large Language Models with contextualised prompts and systematically excluding erroneous code segments., this approach endeavours to streamline the error rectification process and enhance the overall quality of the generated test suites.

Error Handling Methodology

The generated test cases may not always compile due to inherent errors within the code. To address these errors comprehensively, a multi-faceted approach is adopted, which can be broken down into three overarching steps.

1. Error Contextualisation and LLM Reprompting
2. Exclusion of Erroneous Lines and Methods
3. Exclusion of Erroneous Files

Error Contextualisation and LLM Reprompting

The first step in handling the errors is by re-prompting them to the Large Language Model. In this step, the aim is to provide the Large Language Model with more context about these errors. Given that there is a token limit, a more greedy approach is taken in selecting which errors gets more context and which ones do not. To be more precise, we will only aim to add more context to the more commonly occurring errors.

Before we discuss the reprompting methodology, it is imperative to mention a limitation of using the OpenAI API. Unlike the use of the chatgpt model via the web interface, the OpenAI API does not possess any memory capabilities. This means that the model will only utilise the information provided in the prompt. This limits the amount of context that can be given to the Large Language Model, which limits this setup to exclude some details such as the Class Under Test (CUT), instead taking the approach of including the failing test class in the prompt as well as the errors resulting from it with provided context about them.

This Reprompting step is conducted for each project at a time. This is also due to a limitation, which will be discussed in more detail in the implementation chapter, where in a bid to add contextual information such as class signatures, we utilise a library which requires a Java Virtual Machine (JVM) instance which cannot be stopped and restarted in one run, and given the need to provide it a class path for each project, this limits us to operate on an individual project basis.

There are three approaches that can be taken in handling the errors for a project. For each failing test in a project, we can:

1. provide the failing test and one error at a time
2. provide the failing test with multiple errors of the same type at once in isolation
3. provide the failing test with multiple varied errors

For this report, we have chosen the third option as being more efficient, given that providing varied errors can enable the model to tackle multiple errors at once. Had we chosen the other approaches in which the model is required to handle the errors in isolation, it may limit the model's understanding of the broader context and interactions between the different error types.

With this made clear, the reprompting of errors for each project is done in the following phases:

1. Error Logging and Extraction

The first phase involves the Error Logger and Error Finder from the Error Analysis Setup being used to extract any errors that occur within the project. This returns to an array of Error objects for a project.

2. Error sorting

Next, the errors for a project are parsed to a "handle_all_errors_for_project" method within a Reprompter class. These, errors are then sorted into the files that they are located in by grouping them based on their 'file' property. Subsequently, each file and its errors are parsed to a "handle_all_errors_for_file" method in the next phase for handling

3. Error Contextualisation

In the effort to re-prompt the model, this setup will seek to provide the model with instructions to abide by as well as contextual information about the errors. To do so, we will use the following base prompt:

```
Fix the errors in the provided test class:

*test class goes here*

Encountered errors:

*errors with context go here*

Your task is to address the errors identified above while preserving
as much of the test functionality as possible. Ensure that you try to
adhere to this instruction and make only the necessary corrections to
resolve the errors. The resultant test class must be compilable and
achieve high-quality testing, this is the main goal. You must reply
only with the test class code with the corrections.
```

In the course of error contextualisation, a distinct methodology is employed for each encountered error. This categorisation is necessitated by the pragmatic constraints posed by the token limit, which compels us to prioritise the provision of more contextual information for the most prevalent errors. These predominant errors will be subject to a detailed discussion in the implementation chapter, providing a comprehensive understanding of their resolution mechanisms. Conversely, less frequent and unique errors are addressed through a more generalised approach, as it will be discuss shortly. In essence, while common errors benefit from tailored treatment aimed at enhancing comprehension within the constraints of token limitations, rarer errors are managed through a holistic approach, which will be the current focus of this subsection. These errors are handled as follows:

1. A preliminary phrase is used to categorise these errors as less specific and of different types "The following errors are unspecified:"

2. Then for each error that falls in this category, the error message, line, and more details about it (which is provided by the java compiler) are listed:

   Error Type

   Error Line

   Error Details

   The following example showcases a prompt for a class that contains such an error

---

```
Fix the errors in the provided test class:

public class SomeTest {
  Calc calculator = Calculator();


    @Test
  public void testaddition(Int num1, Int num2) {
      assert(5, calculator.compute(1,'4','+');
  }
}
```

```
Encountered errors:

The following errors are unspecified:

Error Type: Syntax Error
Line: 6
Error Details:
[javac]    assert(5, calculator.compute(1,'4','+');
[javac]                                             ^

Your task is to address the errors identified above while preserving as
much of the test functionality as possible. Ensure that you try to adhere
to this instruction and make only the necessary corrections to resolve the
errors. The resultant test class must be compilable and achieve high-
quality testing, this is the main goal. You must reply only with the test
class code with the corrections.
```

---

4. Reprompting

Once the prompt has been generated, the model is re-prompted and the resultant class is saved to the file.

Operational constraints imposed by a token limit often render certain files susceptible to containing an excessive number of errors, surpassing the capacity of a single prompt. In response to such scenarios, the implemented strategy involves systematic error management techniques. This involves maintaining a record of errors incorporated into the prompt in a variable named "handled_errors" , alongside a dedicated variable termed "result_class" initially set to represent the code within the test class. Additionally, the array "handled_errors" is initially devoid of entries. Prior to appending an error to the prompt, a check is performed to determine whether the inclusion of the error would remain within the confines of the token limit, facilitated by the TokenCounter class. If the limit is not exceeded, the error is appended to both the prompt and the handled errors array; otherwise, further additions are precluded. Subsequent to the saturation the prompt, it is submitted to the LLM, yielding a modified result_class value. This iterative process continues, progressively appending unhandled errors to the prompt until all errors have been addressed, as signified by the equivalence of the handled_errors array to the

original file errors. Upon the successful handling of all errors, the last returned class (i.e., the latest value of result_class) is persistently stored to file.

This process entails how errors are handled for each file within a project. For each project, this setup will aim to perform this Error Contextualisation and LLM Reprompting step three times. If some errors are not eradicated, then we move on to the next step.

Exclusion of Erroneous Lines and Methods

In instances where errors persist despite LLM reprompting, the systematic exclusion of erroneous code segments is initiated. This is done in two ways:

1. Exclusion of lines of code that will not propagate into more errors

In this case, the setup aims to remove a line of code from the test class. This is only performed when a line is guaranteed not to propagate into more errors within that test class. Several methods are available to identify lines that are likely to cause errors, such as statically analysing the code structure and examining how lines interact with each other. However, for the purposes of this report, our focus is narrowed to the exclusion of lines associated with assertions. This deliberate limitation is founded upon the premise that such lines, by nature, do not engender error propagation, and constitute the majority of removable lines, as it will be discussed within the implementation chapter.

2. Excluding methods that will not propagate into more errors

In the remaining errors that cannot be handled by removing the lines, this setup takes the approach of excluding the method in which this error is located, if applicable, using the following regular expression:

```
(r'((public|private|protected|static|final|\s)*)\s+[\w<>,\[\]]+\s+(\w+)\s*\
([^)]*\)\s*{([^{}]*)}')
```

While more efficient methods, such as developing a dedicated parser, could potentially yield better results, dedicating resources and time to such an endeavour may not significantly improve outcomes beyond what the regular expression achieves. Moreover, Java's complexity presents a challenge, as crafting a parser capable of handling all scenarios is a complex undertaking. Hence, the approach of this setup prioritises practicality, aiming to address the majority of cases effectively rather than exhaustively. This regular expression method offers a pragmatic solution that satisfactorily addresses the complexities of Java code structures. Having said this, the following method pseudo code is used to match method declarations and get the start and end lines of a method:

```
function get_method_lines_for_line(java_file_path, line_number):
    open java_file_path
    read file content into source_code

    method_pattern = r'((public|private|protected|static|final|\s)*)\s+
    [\w<>,\[\]]+\s+(\w+)\s*\([^)]*\)\s*{([^{}]*)}'

    for each match in find_matches(method_pattern, source_code):
        method_start = count_newlines(source_code, 0, start_of(match)) + 2
        method_end = count_newlines(source_code, 0, end_of(match)) + 1

        if line_number is between method_start and method_end:
            return (method_name_from_match(match), method_start, method_end)

    return None
```

Here is a breakdown of the regular expression:

`((public|private|protected|static|final|\s)*)`

This part captures the access modifiers (public, private, protected), modifiers (static, final), or whitespace characters `(\s)` preceding a method signature. The * quantifier means zero or more occurrences of the enclosed group.

`\s+`

This matches one or more whitespace characters (spaces, tabs, line breaks).

`[\w<>,\[\]]+|:`

This matches the return type of the method. It includes word characters `(\w)` which are typically letters, digits, or underscores, as well as angle brackets (¡¿), commas (,), and square brackets ([]). This part allows for generics and arrays in the return type.

`\s|+`

This matches one or more whitespace characters.

`(\w+)`

This captures the method name, consisting of one or more word characters.

`\s*`

This matches zero or more whitespace characters.

`\([^)]*\)`

This matches the parameter list enclosed in parentheses. `\( ... \)` is used to escape the parentheses. `[^)]*` matches zero or more characters that are not a closing parenthesis).

`\s*`

This matches zero or more whitespace characters.

`{([^{}]*)}`

This captures the body of the method enclosed in curly braces . `[^{}]` matches any character except and , and * matches zero or more occurrences of these characters.

It is imperative to make clear that this setup only exclude methods that will not propagate into more errors. In particular, it only excludes test methods, but not setup (with @Before annotation) or after (with @After annotation) methods. This step is performed 5 times, and any persistent errors are moved on to the next step.

Exclusion of Erroneous Files

In the concluding phase of the error rectification process, any files found to contain errors are systematically excluded. These instances typically involve errors embedded within the imports, setup, and after methods of test classes, which are beyond the remedial capacity of the LLM or cannot be mitigated through line or method exclusions. Consequently, all files exhibiting such persistent errors are systematically removed from the project until the entirety of the identified issues is resolved

Conclusion

The Error Ratification Setup establishes a structured framework to comprehensively address compilation errors encountered during the generation of test cases. By reprompting the LLMs with contextualised prompts and systematically excluding erroneous code segments, this approach aims to eradicate errors effectively, ensuring the generation of high-quality, compilable test suites. It is imperative to take note that each time errors are handled, The error analysis setup is employed to log and extract errors again in anticipation of changes to the errors. To provide a final overview of the three steps that constitute this setup, we have provided the following pseudocode:

```
function reprompt_for_project(project_path):

    logger = create CompilationErrorLogger(directory)
    error_finder = create ErrorFinder(directory)

    # the errors to be handled
    errors = error_finder.find_errors_for_project(project_path)

    #sort errors into files
    files = {}
    for each error in errors:
        if error.file in files:
            files[error.file] += 1
        else:
            files[error.file] = 1

    # Number of rounds the LLM is reprompted
    reprompt_rounds = 3

    # Number of rounds of line and method exclusion
    line_removal_trials = 5

    total_deleted_files = 0
```

```
# Loop until there are no more errors in the project
while length(errors) > 0:
    logger.clean_project(project_path)
    logger.log_compilation_errors_for_project(project_path)
    error_finder = create ErrorFinder(directory)
    errors = error_finder.find_errors_for_project(project_path)

        # Create reprompter for method
    reprompter = create Reprompter(errors, api_key)

        # Step 1: Error Contextualisation and LLM Reprompting
    if reprompt_rounds > 0:
        reprompt_rounds -= 1
        reprompter.handle_all_errors_for_project()

    # Step 2: Exclusion of Erroneous Lines and Methods
    elif line_removal_trials > 0:
        line_removal_trials -= 1
        reprompter.exclude_errors(errors)

    # Step 3: Exclusion of Erroneous Files
    else:
        deleted_files = reprompter.exclude_error_files(errors)
        total_deleted_files += deleted_files

#Final Statistics
print error_finder.generate_statistics(errors)
```

Conclusion

The Error Reprompting Setup establishes a structured framework to comprehensively address compilation errors encountered during the generation of test cases. By reprompting the LLMs with contextualised prompts and systematically excluding erroneous code segments, this approach aims to eradicate errors effectively, ensuring the generation of high-quality, compilable

test suites.

5.3.9 Results Measurement Setup

The Results Measurement Setup aims at assessing the quality and effectiveness of the generated test suites. This subsection delineates the methodology employed for measuring the metrics discussed earlier in this report, including coverage, mutation score and the code style of the generated test suites.

The evaluation process commences following the Error Rectification Setup, wherein all compilation errors are addressed, ensuring the generation of error-free test suites. Subsequently, this stage facilitates the comprehensive assessment of the generated test suites based on the predefined criteria.

Coverage Measurement

The measurement of coverage serves as a fundamental aspect of evaluating the effectiveness of test suites. For this purpose, Jacoco [38], a widely used coverage measurement tool for Java, is employed. This report will assess coverage in terms of line (statement) and branch coverage. Each project in the evaluation dataset is configured with the necessary jacoco JAR to facilitate coverage measurement. Subsequently, ant tasks are configured for each test suite to measure coverage.

Jacoco operates in three stages in a bid to produce coverage statistics, which are outlines below:

1. Instrumentation: The compiled test files are instrumented by a Jacoco agent to generate coverage data. Here Jacoco creates a new set of instrumented tests, whereby the bytecode of each of these test files is supplemented with data to assist in the measurement of coverage.

2. Test Execution: The instrumented test files are executed using JUnit, ensuring the execution of the test cases. During the running of these tests, the Jacoco agent captures coverage data and saves them into an executable file.

3. Coverage Report Generation: Upon test execution, Jacoco utilises the coverage data captured in the executable file to produce a coverage report.

An ant task is configured for each of these stages. Once coverage reports are generated, a general line and branch coverage statistic for the entire dataset is calculated for the entire dataset by averaging these values for each project in the dataset.

Mutation Score Measurement

The Mutation Score Measurement represents a crucial aspect of evaluating the effectiveness and robustness of the generated test suites. In this subsection, the methodology for measuring mutation score using Pitest [44], a widely acclaimed mutation testing tool for Java, is elucidated.

Mutation testing aims to evaluate the quality of test suites by introducing artificial faults, known as mutations, into the source code and assessing whether the test suite detects these mutations. This report will utilise Pitest, a widely acclaimed mutation testing tool for Java. Pitest automates the mutation process by generating mutants and running the test suite against them, thereby measuring the mutation score, which indicates the effectiveness of the test suite in detecting faults. This is done in the following steps:

1. Mutant Generation: Pitest generates mutants by introducing faults, such as modifying operators, changing method calls, or altering conditional expressions, into the source code.

2. Test Execution: The test suite is executed against the mutated source code, aiming to detect and fail tests for mutants that indicate potential faults.

3. Mutation Score Calculation: Pitest computes the mutation score, representing the percentage of mutants killed by the test suite compared to the total number of mutants generated.

An Ant task is configured for each package within the project to measure mutation score and generate a detailed report. This approach mitigates resource constraints, as mutation testing can require significantly more resources than traditional testing methods.

Once mutation score reports are generated for each package, a general mutation score is calculated for each test suite by averaging the scores obtained within each package. This aggregated score provides a comprehensive assessment of the test suite's effectiveness in detecting faults across the entire dataset.

Code Style Measurement

The Code Style Measurement segment of the setup involves utilising Checkstyle [9], a renowned tool for enforcing coding conventions, to assess the adherence of the generated test suites to predefined coding standards. This subsection outlines the process of checking the test suites against two prominent style guides: the Google Java Style Guide [19] and the Sun Java Style Guide [37].

Checkstyle serves as a powerful tool to ensure code consistency and maintainability by enforcing coding conventions. This setup will be utilising Checkstyle in the following steps:

1. Configuration: Ant tasks are created for each test suite to execute Checkstyle and analyze

the code against the predefined style guides.

2. Style Guide Evaluation: Checkstyle examines the test suite codebase and identifies violations of the specified coding conventions outlined in the Google Java Style Guide and the Sun Java Style Guide.

3. Violation Reporting: Checkstyle generates comprehensive reports detailing the detected violations, providing insights into areas where the test suites diverge from the prescribed coding standards.

The generated reports from Checkstyle serve as the basis for deriving graphical illustrations of the code style violations. These illustrations provide a visual representation of the extent and distribution of violations across the test suites, as it will be demonstrated in the evaluation chapter.

5.3.10 Wrap up

The Design Chapter sets the stage for the evaluation of test generation techniques employing Large Language Models (LLMs) in software testing. It delineates the framework for experimental setup, from dataset selection to tool configuration, elucidating the methodology for test generation, error analysis, and evaluation metrics. Each subsection offers a detailed exploration of the setup, emphasising the meticulous planning and execution essential for robust evaluation. As this report transitions to the Implementation Chapter, the groundwork laid in this section provides the foundation for implementing and refining the proposed methodologies. Through the subsequent chapters, this report will delve deeper into the practical application of these techniques, offering insights into the efficacy and potential implications of the testing techniques outlined in the report.

Chapter 6

Implementation

6.1 Overview

The Implementation Chapter delves into the practical execution of the methodologies outlined in the preceding Design Chapter. It encapsulates a comprehensive series of steps aimed at achieving the overarching objectives set forth in the Requirements section. This chapter unfolds with the process of Prompt Generation, marking a departure from the sequence presented in the Design Chapter. This strategic shift is orchestrated to ensure that the optimised prompt is used to fine tune the model during the subsequent Fine Tuning phase, thereby aligning the fine-tuned model's behaviour more closely with the intended test generation process.

Following the Fine Tuning of the model phase, the chapter branches into three distinct streams: Evosuite Test Generation, and Test Generation using both Untrained and Trained Models. Furthermore, the chapter encompasses Error Analysis in which the errors generated by the trained and untrained models are again analysed and Error Ratification which seeks to rectify these errors within the generated test suites. Lastly, the chapter culminates in Results Measurement, where the efficacy and performance of the generated test suites are rigorously evaluated using the established metrics and tools discussed in the previous chapter. Through this comprehensive implementation, the chapter aims to achieve the overarching objective laid out in the Requirements Section.

6.2 Prompt Generation

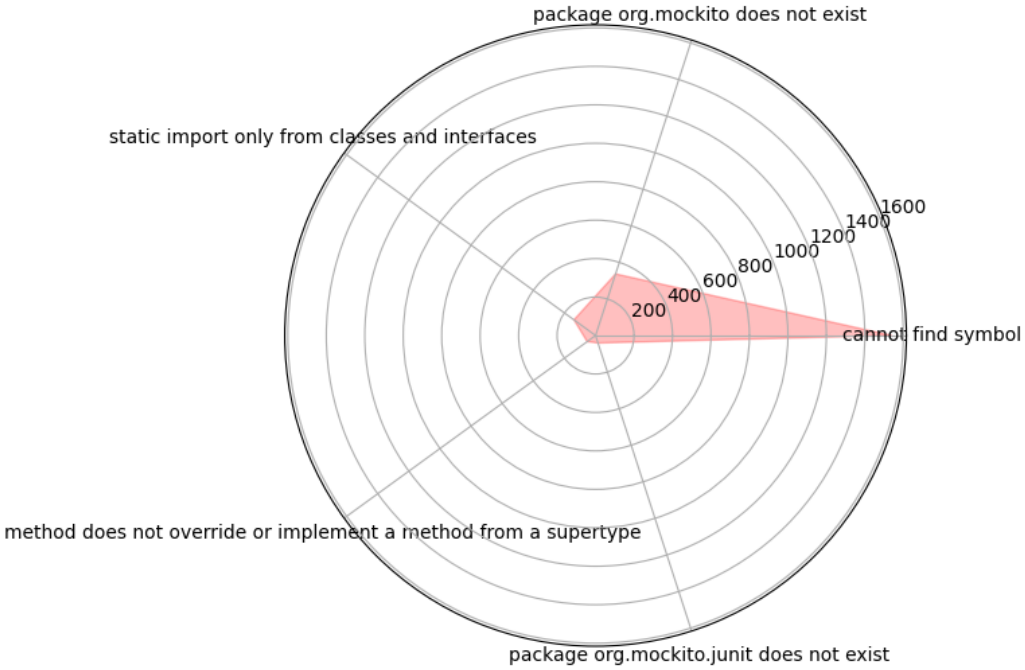
The commencement of the Prompt Generation phase entails leveraging the established framework from the preceding Test Generation phase to curate an initial test suite for each project within the dataset using the gpt-3.5-turbo model. As expounded upon in the design chapter, this process initiates with the creation of a dedicated directory, termed 'untrained_tests,' designed to house all generated tests for a given project.

As delineated in the preceding chapter, each '.java' file undergoes a process of code extraction, subsequent to which it is appended to a JSON conversational template, preparing it for prompt submission to the Large Language Model (LLM). The initial prompt utilised in this phase is structured as follows:

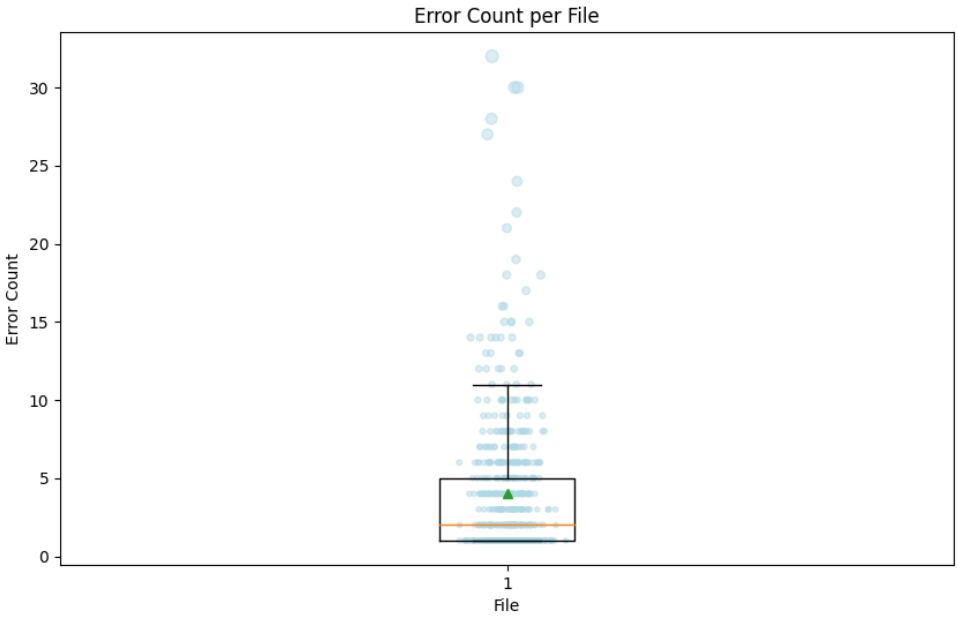
'Generate a Java test class using JUnit4 for the following Java class, and reply with code only:'

Subsequently, the API response containing the generated code is extracted and written to the corresponding test file. This iterative process is applied to all eligible Java files within the source directory, ensuring adherence to token constraints. Employing this systematic approach, a total of 3,143 tests were generated across the evaluation dataset, with a cumulative token query count of 11,612,4111. However, 149 files could not be prompted due to exceeding the token limit.

In the subsequent step, error extraction is undertaken utilising the ErrorLogger and ErrorFinder classes, as elucidated in the Error Analysis setup within the design chapter. The generated tests are compiled for each project in the evaluation dataset, with any standard output (stdout) being logged into a text file within the project base directory. The ErrorFinder then analyzes this text file and generates a list of Error objects based on any detected errors. Initial statistics generated by the ErrorFinder reveal a total of 2,383 errors, with 'cannot find symbol' and 'package SomePackage does not exist' errors constituting 82% of the total errors, sa depicted by the adarr plot below.



Furthermore, highlighting the distribution of errors per file, the median number of errors per file was approximately 2, with the maximum and minimum numbers of errors per file being 33 and 1, respectively. This distribution is visually depicted in the accompanying box plot, which provides a clearer illustration of the spread of errors across the dataset.



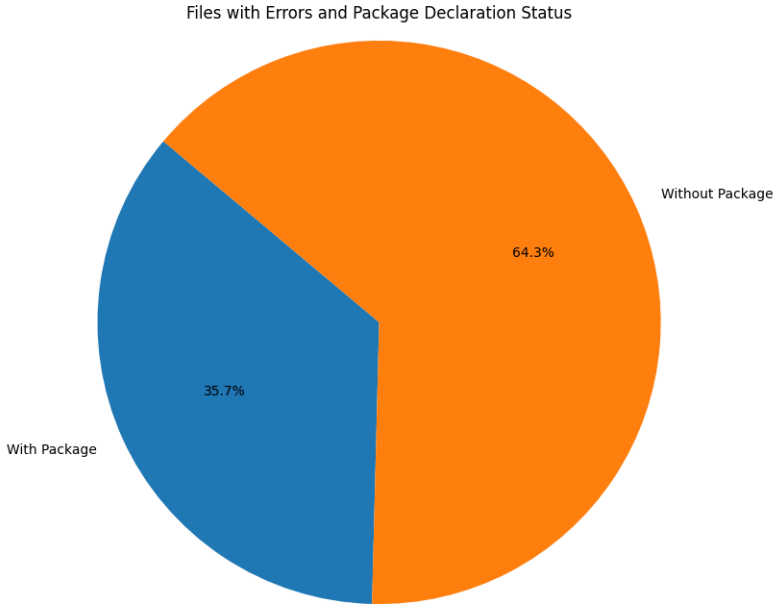
In delving into the statistics surrounding each error type, particularly the 'cannot find symbol' errors and 'package SomePackage does not exist' packages, this section aims to unravel the underlying causes of these prevalent issues. Through meticulous analysis, it will consider three instances. Firstly, it discern whether these errors stem from direct missing dependencies within the Ant compilation task during the experimental setup, or if the errors are indicative of broader challenges within the project structure. Subsequently, if the prior reasoning appears unsatisfactory, this exploration will focus on dissecting the root causes of the 'cannot find symbol' errors, probing into potential issues such as undeclared variables or incorrect import and package declarations. Following this investigation, we will pivot towards examining the causes of the 'package SomePackage does not exist' errors, aiming to elucidate any deficiencies in the use of incorrect dependencies. By delving into these intricacies, we aim to provide a comprehensive understanding of the factors contributing to these errors and pave the way for targeted solutions in the subsequent sections.

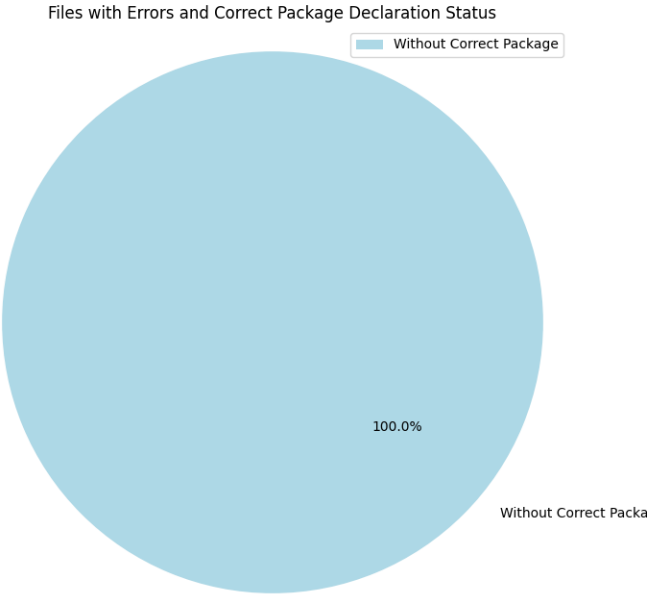
Considering class path issues in the experimental setup

First, an examination of classpath issues within the experimental setup is considered. As part of our debugging process, the dependencies loaded in the classpath were checked to ensure their accuracy. This investigation revealed that the compiled source and test files, along with any libraries within the project's 'lib' directory, were indeed loaded at runtime as expected. This verification affirms that the encountered errors are not attributable to shortcomings within the experimental setup. By ruling out classpath discrepancies, we can confidently focus our attention on other potential sources of error within the project structure.

A deeper examination of the cannot find symbol errors

Thereafter, the analysis delves deeper into the characteristics of the 'cannot find symbol' errors. In this context, particular attention is directed towards examining the presence and correctness of package declarations within files containing these errors. Notably, two graphs were generated to elucidate these aspects: the first revealing that 60



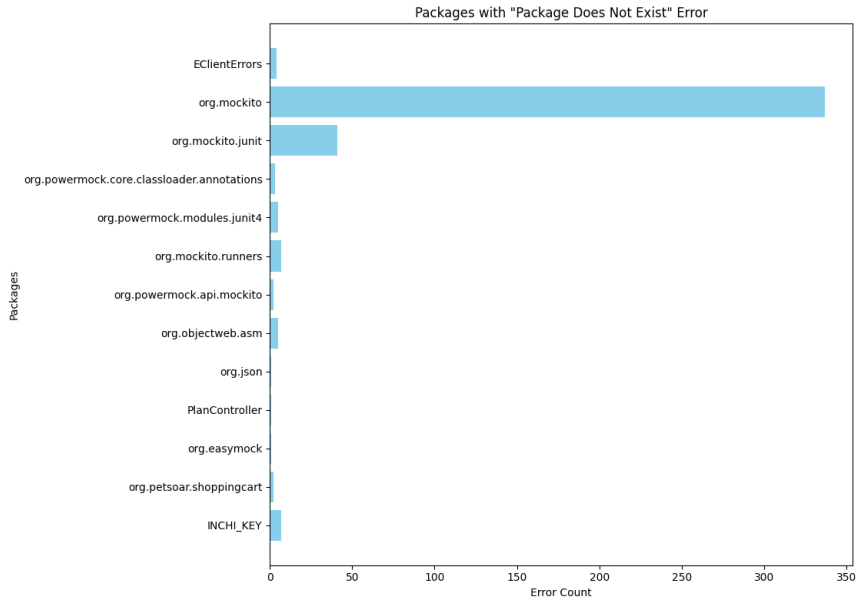


This observation aligns with the absence of contextual information provided to the Large Language Model during the test generation phase regarding the package location of the file under consideration. To substantiate this inference, correct package declarations were subsequently added to the files, resulting in a reduction of total errors from 2,415 to 1,556. Specif-

ically, the 'cannot find symbol' errors decreased from 1,808 to 1,021, constituting a notable decline of 43.5%. The persistence of remaining 'cannot find' errors suggests potential idiosyncrasies specific to each of the tests generated by the Large Language Model.

A deeper analysis of the "package SomePackage does not exist" errors

Subsequently, a more comprehensive examination is conducted to elucidate the underlying causes of the "package SomePackage does not exist" errors. Initially, a bar graph was generated to illustrate the distribution of packages responsible for triggering these errors. This is demonstrated below:



Remarkably, the graph delineates that all implicated packages were absent from the classpath, as they were not initially deemed requisite. These packages predominantly represent libraries integrated by the Large Language Model (LLM) during testing, which were not originally included in the project dependencies. Notably, among these packages, Mockito emerges as the most prevalent.

To mitigate such occurrences, a shared 'lib' directory is incorporated into the classpaths of all evaluation projects. Within this shared 'lib,' only approved libraries that the LLM is authorised to utilise are included, along side the dependencies of each project. These sanctioned libraries include JUnit, Hamcrest, and Mockito. Furthermore, the persistence of these errors underscores the absence of guidance provided to the LLM regarding the selection of dependencies, thereby necessitating its inclusion of any libraries deemed essential for test generation.

An investigation into the "class Someclass is public, should be declared in a file named SomeClass.java" errors

Although not as prevalent as the missing symbol or dependency errors examined previously, the "class Someclass is public, should be declared in a file named SomeClass.java" seem to provide more insightful findings that can guide this report's prompt generation undertaking. Upon closer examination of these errors, it becomes evident that they all stem from inconsistencies between class names and file names. According to Java convention, a file must be named after the class it contains. During our test generation phase, files were named after the Class Under Test (CUT) suffixed with 'Test'. However, the test classes generated by the Large Language Model (LLM) in these errors appear to have deviated from this convention. Variations in names, such as the CUT name prefixed with 'Test' or entirely different class names, were observed. This discrepancy underscores the absence of specific instructions provided to the LLM regarding the naming of test classes, resulting in deviations from expected naming conventions. Furthermore, it is noteworthy that some file names appeared completely dubious, suggesting that test files were generated for files that were not conducive to testing. In the setup for the model test generation it was assumed that every file was testable and contained a class to test. However, this assumption proved inaccurate in cases such as package-info.java files, where test classes were created with file names like package-infoTest.java, which lacked coherence or any classes within them. To address this issue, a refinement of the test generation setup is conducted by implementing a mechanism to exclude files that do not contain clear class declarations, thereby ensuring more accurate and coherent test generation.

In light of our examinations, several key components emerge as essential in refining our prompt generation process. Firstly, specifying a consistent naming convention for the test class, aligning it with the Class Under Test (CUT) name suffixed with 'Test', ensures adherence to Java conventions. Secondly, including a package declaration within the generated test files not only reinforces organisational structure but also mitigates 'package does not exist' errors. Lastly, delineating a predefined set of dependencies that the Large Language Model (LLM) is permitted to utilize, encompassing libraries such as JUnit, Hamcrest, and Mockito as well as specific project dependencies, fosters consistency and compatibility across generated tests.

To ensure the precise specification of the test class package, a method leveraging regular expressions is proposed. This method facilitates the extraction of the package containing a file from its file path, thereby aligning the generated test class with its corresponding package. Similarly, the test class name is obtained by extracting the Class Under Test (CUT) name from the

file name and appending the suffix 'Test', in accordance with established naming conventions. Furthermore, to compile a comprehensive list of dependencies, an iterative traversal of both the project's lib directory and the shared 'lib' directory is undertaken. Subsequently, the names of JAR files located within these directories are extracted, thereby encompassing all requisite dependencies for the test generation process. With this methodology, the final prompt for this phase is as follows:

```
You are a Java software developer tasked with thoroughly testing a
provided Class Under Test using the JUnit 4 testing framework. This Class
Under Test is located within the package {package}.
To begin, ensure that you create a test class in Java 11 (JDK 11) with a
specific name: {class_name}. This name is essential for consistency and
clarity.
Next, let's focus on the dependencies. Import only the necessary classes
and libraries from the provided dependencies string: {dependencies}. It's
crucial to avoid importing any libraries or dependencies outside of this
list.
As you delve into writing the test methods, remember to strictly test the
public interface (methods) of the provided class {contents}. Do not test
private methods or abstract classes, maintaining a focus on the public API.
Your task is to write well-documented and appropriately named test methods
that cover all aspects of the Class Under Test's functionality. Each test
method should be documented, named sensibly, and include relevant
assertions to validate the behavior.
Ensure that the test class includes a setup function that creates all
necessary objects required for testing. This setup is vital for
initializing the environment before each test method execution.
Additionally, it's imperative to include a package declaration at the top
of the test class file to prevent compilation errors. Use the provided
package name: {package}.
Lastly, aim for comprehensive test coverage, including edge cases,
boundary conditions, and typical scenarios. Your goal is to provide
thorough validation of the Class Under Test's behavior.
```

Now, with all the guidelines in place, proceed to generate the test class based on the provided contents:

6.3 Fine Tuning

Fine-tuning of the gpt-3.5-turbo-0125 model, as detailed in the design chapter, proceeded through a meticulously structured approach encompassing three distinct steps.

Data Collection

Data procurement commenced with the gathering of information from six open-source projects, as outlined in the design chapter. To facilitate this process, a dedicated 'DataSources' directory was established, housing individual directories corresponding to each project. Within each project directory, 'src' directories containing 'main' and 'test' subdirectories were created. Project repositories were obtained in ZIP format from GitHub. Given the inherent diversity in project structures, manual navigation through the repository structures was necessary to locate and extract the source and test directories. Subsequently, these directories were copied into the respective 'src' and 'test' directories for each project within the 'DataSources' directory.

Following this procedure, each project adopted a uniform format for source and test directories, thereby enabling traversal with a standardised algorithm. Subsequent to the data collection phase, a 'TestCollector' class was employed. For each project within the 'DataSources' directory, this class traversed the test directory ('src/test') and identified each '.java' file. For each discovered file, path manipulation was performed by removing 'Test' from the filename and replacing 'src/test' with 'src/main'. The resulting path was then verified for existence. If confirmed, a test pair was established. These pairs were recorded as tuples in an array. To store the pairs from this initial data collection phase, a directory named 'Stage_One_Tests' was created. Within this directory, individual directories were generated, each corresponding to an index of the tuple array. Within each directory, the focal class and test class were copied and renamed as 'focal_class.java' and 'test_class.java', respectively. Additionally, a JSON file containing a dictionary with values for the package, class name, and project of origin for the test class was included. This concluded the data collection phase, yielding a total of 1659 pairs collected from the aforementioned projects.

Data Preparation

In this phase, the collected test pairs from the previous data collection step undergo formatting for further processing. The formatted tests are stored in a designated directory named 'Stage_Two_Tests'. For each pair obtained in the initial stage, the test and focal class code are extracted. Additionally, any commented metadata commonly found at the beginning of the file are removed to streamline the code. Furthermore, to enrich this code, essential information such as the package, class name, and dependencies for each test class are incorporated. The package and class name were previously saved in a JSON file during the data collection step, along with the name of the project that the pair is from. As for the dependencies, these were predetermined by analysing the 'pom.xml' and 'build.gradle' files of the respective projects to identify the necessary dependencies for testing capabilities. Subsequently, these dependencies were compiled into a dictionary format. Given that each pair's project can be determined from the JSON file from the data collection stage, the dependencies for a test pair are retrieved by querying this dictionary.

With this structured setup, the prompt generated from the prompt generation step is contextualised with the package, class name, and dependencies for the focal class, and serves as the template for the user role. The resulting test class is tailored to the assistance role, and these two are then incorporated into a conversational chat-style JSON format. This formatted JSON file is saved within the 'Stage_Two_Tests' directory, with the respective index number serving as the file name. This concludes the data preparation phase.

Fine Tuning

In this phase, the data extracted from 'Stage_Two_Tests' is loaded into an array. Given the extensive dataset consisting of 1659 test pairs, it is imperative to downsize to a manageable size for fine-tuning purposes. Therefore, a subset of 250 test pairs is randomly selected from the formatted data. To ensure adherence to token constraints, a token counter is employed during the selection process, ensuring that none of the chosen JSON files surpasses the token limit.

Once the 250 JSON files are selected, they are compiled into a single JSONL file format. Subsequently, this compiled file is uploaded to the OpenAI platform for fine-tuning. During this phase, token and cost estimation are conducted to optimize resource allocation.

Upon successful upload, a fine-tuning job is initiated on the gpt-3.5-turbo-0125 model, with the selected dataset. The model undergoes training for three epochs, each comprising a batch size of 1. The fine-tuning process culminates in the generation of various statistics, with a

training loss of 0.1028 having been trained on 2,421,582 tokens.

6.4 Evosuite Test Generation

Before proceeding with the Evosuite test generation, it's imperative to compile the main source code for each project within the dataset. This compilation process is executed via the command line interface, resulting in the creation of a JAR file containing the compiled classes. This JAR file is generated within the project's base directory using the following command:

```
jar cf *project_name*.jar build/classes
```

Additionally, in the shared 'lib' directory for all projects, a Evosuite (version 1.2.0) JAR is uploaded, serving as the primary tool for test generation. Following this setup, the properties required for the Evosuite test generation are configured using the command line interface:

```
java -jar ../lib/evosuite-1.20.0.jar -setup -cp lib/* <<project-name>>.jar
```

This command establishes the classpath, including all dependencies within the project's 'lib' directory, along with the JAR file that was earlier created in the project base directory. Default Evosuite settings are utilised during this setup.

Subsequently, Evosuite tests are generated for each project within the dataset. A search budget (timeout) of 2 minutes is allocated for this process. The following command is executed to initiate test suite generation:

```
java -jar ../lib/evosuite-1.2.0.jar -target <<project-name>>.jar Dsearch_budget=120
```

This command specifies the target JAR file and sets the search budget to 120 seconds. Upon execution, Evosuite test suites are successfully generated for every dataset in the evaluation dataset

6.5 Large Language Model Test Generation

Aligned with the methodologies delineated in the design chapter, the Large Language Model (LLM) test generation process encompasses the creation of two distinct test suites: one employing the ordinary model and the other leveraging the fine-tuned model. Both test suites are generated with the same setup but prompting different models and storing the resultant tests in different directories.

An enhancement to the test generation setup was introduced following to the investigation into errors resulting from mismatches between test class and test file names in the prompt

generation setup, where some files were found to be untestable. This enhancement involves the modification of the test generation criteria to exclusively create tests for files deemed 'testable,' defined as those containing at least one class declaration. This condition was facilitated by the utilisation of the following regular expression:

```
r'\s*(public|private)?\s*(abstract)?\s*class\s+(\w+)\s+((extends\s+\w+)|
(implements\s+\w+(  ,\w+)*))?\s*{'
```

This regular expression seeks to match Java class declarations within source code files. It scans for lines containing class declarations, by matching lines that contain modifiers (e.g., 'public', 'private'), optional keywords (e.g., 'abstract'), and the class name itself. Additionally, it accounts for class inheritance and implementation clauses. This refined matching criteria facilitates the generation of tests for files with viable Java class structures

Following the incorporation of this regex, an initial untrained test suite was generated utilising the ordinary model. However, upon inspection, it was observed that the resultant test suite exhibited a substantial reduction in size than expected. This diminution was attributed to the regex's inclusion of the '' symbol at the end, causing it to match only files in which the opening bracket appeared on the same line as the class declaration. Consequently, lines with the opening bracket in the next line were excluded from this matching process. To rectify this discrepancy, the opening bracket was removed from the regular, allowing for a comprehensive matching process. After this„ the missed tests were regenerated, culminating in a total of 16,124,111 tokens queried to the model and yielding 3138 test classes in total.

An analogous approach was adopted for the fine-tuned model, resulting in the generation of a commensurate number of test classes and the utilisation of a similar token count. The resultant test suites were segregated into the 'trained_tests' and 'untrained_tests' directories for the fine-tuned and ordinary models, respectively, across every project within the evaluation dataset. This encapsulates the proceedings of this subsection.

6.6 Untrained Error Analysis

Before examining the analysis of errors stemming from the test suite generated by the ordinary model, it is imperative to acknowledge a critical caveat pertaining to the figures obtained during this analysis. It was observed in the latter stages of this evaluation that the figures obtained in this step were incomplete but nevertheless provided a snapshot of the overall error landscape accurately. This discrepancy arose due to the default behaviour of the Java compiler (javac),

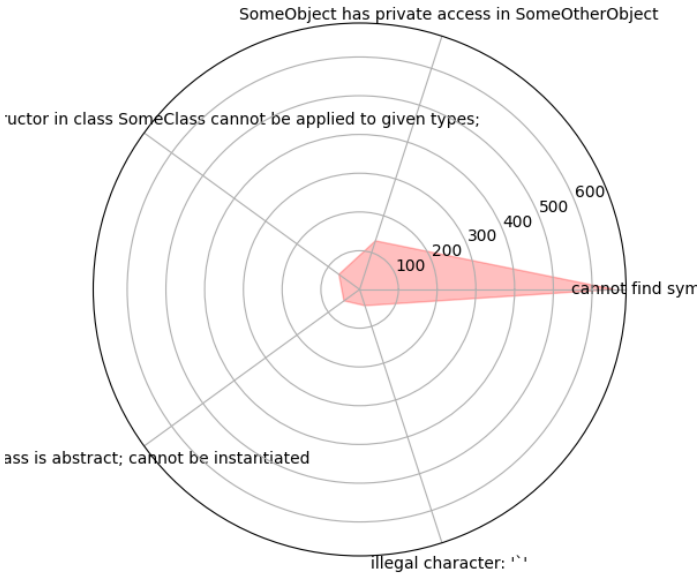
which limits the number of errors it throws to only 100 per project. To address this issue and ensure the comprehensive capture of all errors, the 'xmaxerrs' argument was incorporated into the Ant tasks responsible for compiling the tests. The argument was set as follows:

```
<compilerarg value="-Xmaxerrs"/>
<compilerarg value="0"/>
```

This addition instructs the Java compiler to report all encountered errors. Upon revisiting the error compilation process post-discovery, it was ascertained that the actual total errors amounted to 13,657 across all projects in the dataset. However, it is noteworthy that despite this revision, the composition among the most common error types remained consistent. Consequently, the figures utilised in the initial error analysis were essentially a downsized representation of the overarching trends observed.

With this aforementioned caveat in mind, it is fit to proceed with the analysis of errors. With the 100-error-per-project limit, the total number of errors aggregated to 1302. Notably, this represents a 45.4% reduction in total errors compared to the 2383 errors generated using the initial prompt in the prompt generation step.

The analysis revealed that the most prevalent errors in the test suites generated by the ordinary model were once again the 'cannot find symbol' errors, totaling 657 occurrences. These errors accounted for approximately 50.4% of the total errors, a notable decrease from their previous dominance constituting 64% of errors during the prompt generation phase. However, 'cannot find symbol' errors still represent a substantial portion of the error landscape. Conversely, package errors saw a decline in significance, comprising only 23 errors (1.7%) compared to their previous prevalence during the prompt generation phase, where they accounted for 17.5% of the total errors (416 in total). It is noteworthy that excluding the 'cannot find symbol' errors, the composition of the top 5 errors has undergone a complete shift. This transformation is visually represented in a radar plot provided below:



Having examined the overall landscape of errors in the test suites generated by the ordinary model, we now delve deeper into the characteristics of specific error types that offer valuable insights. Among these, 'cannot find symbol' errors stand out as a significant category warranting closer scrutiny to start off. By dissecting these errors and understanding their origins and implications, this stage seeks to gain deeper insights to address the error ratification stage. Therefore, it will commence our detailed analysis by focusing on the characteristics and implications of 'cannot find symbol' errors.

Cannot find symbol errors

In Java, 'cannot find symbol' errors primarily arise because the Java compiler needs to verify that all identifiers used in the code are available in the classpath. When the compiler encounters an identifier that it cannot resolve, it throws this error.

Upon careful examination, these errors seem to stem from two distinct situations:

The first situation occurs when the symbol exists within the project source code but is not imported correctly. These symbols are typically located in packages different from the file attempting to use them. Given that the Large Language Model (LLM) was provided with the package information for its Class Under Test (CUT) but not for these external classes it attempts to use, these errors arise

The second situation arises when the symbol is incorrectly spelled or used. This can include

spelling errors, incorrect annotations, or the use of a completely nonexistent symbol.

When the Java compiler throws such an error, it typically provides a structured error message containing the symbol and its location. For example:

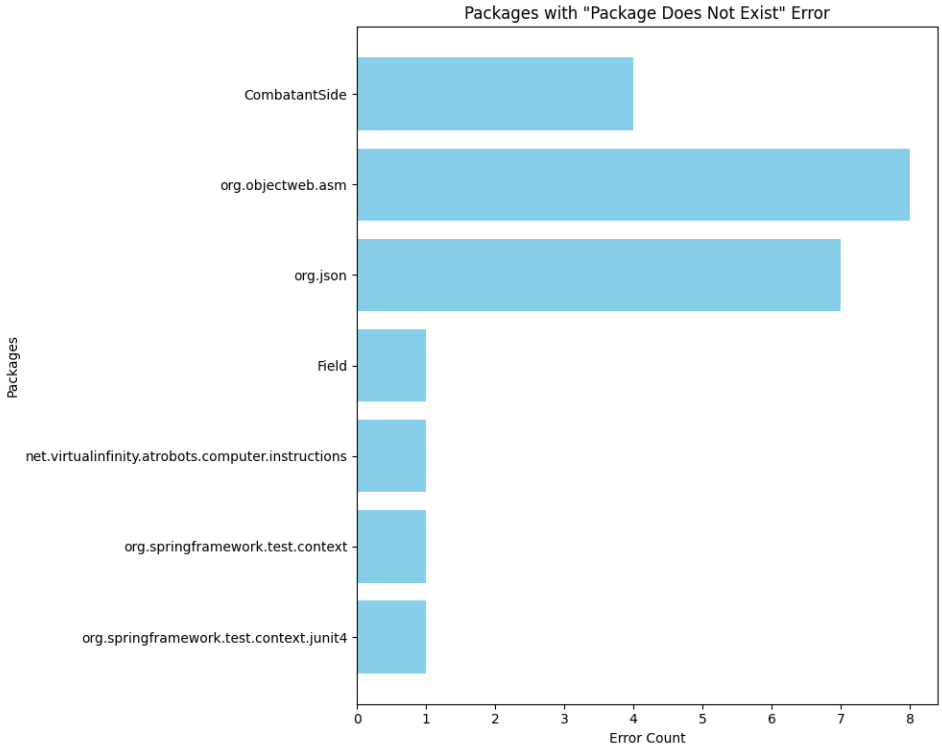
```
[javac] import de.outstare.fortbattleplayer.model.SimpleCombatant;
[javac]                                                 ^
[javac]   symbol:   class SimpleCombatant
[javac]   location: package de.outstare.fortbattleplayer.model
```

This error is an example of the first situation, where the symbol is imported incorrectly. The correct import path should have been de.outstare.fortbattleplayer.model.impl.SimpleCombatantTest.

The Java compiler usually provides the symbol keyword (e.g., class, method, variable, constructor, static) and the name of the symbol in the error message. The most prevalent keywords are 'class,' 'method,' and 'variable.' Errors with the 'method' keyword typically arise from incorrect or unknown method calls, while errors with the 'class' keyword often stem from improper imports. In these errors, the class that the object is an instance of is provided by the compiler along with the keyword. Similarly, errors with the 'variable' keyword also usually arise from incorrect imports, with the name of the class provided in the location parameter of the error details.

package SomePackage does not exist errors

The occurrence of 'package SomePackage does not exist' errors has become less prominent in the test generation process. However, the residual errors appear to stem from instances where the Large Language Model (LLM) utilises a dependency despite being explicitly instructed to use specific dependencies. Additionally, a few instances of these errors arise from static importations. In Java, static importations allow members (fields and methods) defined in a class to be used in Java code without specifying the class explicitly. The following bar graph shows the distribution of the remaining errors.



constructor SomeClassConstructor in class SomeClass cannot be applied to given types errors

An illustrative example of the Java compiler (javac) output for these errors is as follows:

```
constructor Person in class Person cannot be applied to given types;
[javac] Person person = new Person("John", "Doe");
[javac] ^
[javac] required: no arguments
[javac] found: String,String
[javac] reason: actual and formal argument lists differ in length
```

These errors arise when the Large Language Model (LLM) attempts to create an object without possessing sufficient context regarding the constructor of the class it is instantiating. As a result, the LLM makes assumptions about the parameters required for the constructor. The javac output for these errors typically includes the required parameters to create an instance of the class, along with the name of the class to which the object belongs and a reason for the error.

Reached end while parsing errors

Upon examination, errors indicating "Reached end while parsing" typically arise from two distinct situations.

Incomplete Test Classes: One scenario contributing to these errors is when the Large Language Model (LLM) generates an incomplete test class. This often occurs due to the token limit imposed during the model's interaction with the prompt and completion. According to OpenAI's documentation, the maximum tokens allowed for a chat request with the gpt-3.5-turbo model is 16,000, which must be shared between the prompt and completion. Consequently, if the prompt consumes a significant portion of the token limit, the completion may be truncated, resulting in an incomplete test class. Additionally, there is a cap of 4097 tokens on the response generated by the model, further constraining the length of the completion.

Syntax Errors and Missing Closing Symbols: Another situation leading to "Reached end while parsing" errors occurs when the model generates a test class with syntax errors, such as forgetting to include a closing brace '}' symbol. This oversight disrupts the parsing process, causing the parser to reach the end of the file prematurely and consequently triggering the error.

The analysis of errors encountered in the untrained test suite provides valuable insights into the limitations and challenges inherent in the Large Language Model (LLM)-based test generation process. These insights will inform our efforts to enhance the error rectification setup in subsequent sections.

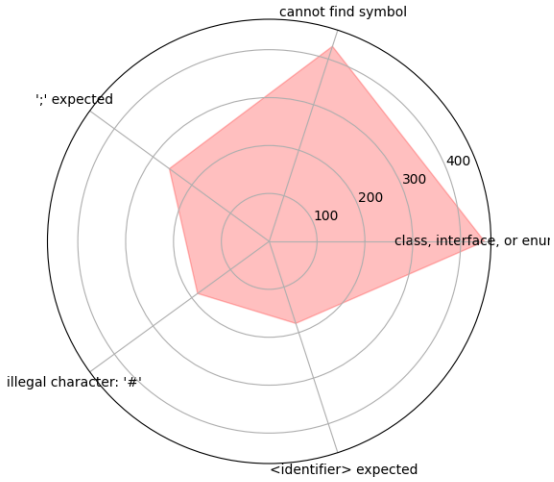
6.7 Trained Error Analysis

The methodology for analysing errors in the trained test suite paralleled that of the untrained error analysis in the preceding section. However, akin to the untrained suite analysis, the figures presented in this section were subject to the constraint of the Java compiler's limitation of 100 errors per project, as mentioned previously. Consequently, while the data obtained may not represent the complete error landscape, it nonetheless offers a snapshot that accurately captures the prevailing trends.

Having established the framework for analysis, it is noteworthy that the total number of errors amounted to approximately 15,629 errors when the constraint of 100 errors per project was removed. This figure represents an increase of 1,972 errors compared to the untrained test suite. However, with the limit imposed by the compiler, the total was reduced to 2,563 errors,

still surpassing the 1,302 errors aggregated in the untrained tests analysis.

In contrast to the untrained suite, the distribution of errors in the trained suite exhibited notable differences, with only the 'cannot find symbol' errors being common in the top five errors of both suites. The most prevalent error type in the trained suite was the 'class, interface, or enum expected' errors, accounting for a total of 450 occurrences. Following closely were the 'cannot find symbol' errors, with 428 instances. Together, these two categories constituted 34% of the total errors. Furthermore, it is worth mentioning that the distribution of errors among error types appeared to be more evenly spread in this test suite compared to the untrained test suite, as illustrated in the radar plot to be presented below.



Subsequently, an examination of errors stemming from the test generation setup revealed a consequential issue. Specifically, in the test generation setup, code extraction relied on isolating the portion of the response enclosed within "‘‘‘" blocks. However, this method assumed that the Large Language Model (LLM) would consistently encapsulate code within these designated blocks, as expected. Regrettably, it was observed on several occasions that this was not the case. Consequently, words were inadvertently included in the Java test files. As a result, the Java compiler (javac) was unable to analyse these files comprehensively, leading to an incomplete enumeration of errors. To address this challenge, a heuristic approach was implemented to eliminate any extraneous text or "‘‘‘" blocks from the extracted code snippets.

6.8 Error Ratification

Moving forward, this chapter transitions into the implementation of the error ratification setup which aims at rectifying prevalent errors encountered during the test generation process. Build-

ing upon the design chapter, this section will expand the setup to encompass specific error types that emerged frequently across the untrained test suite. These errors are:

1. cannot find symbol
2. package SomePackage does not exist
3. constructor SomeClassConstructor in class SomeClass cannot be applied to given types,
4. reached end of file while parsing

To enhance efficacy, a duplicate of the untrained test suite was created and experiments are conducted on it utilising contextual information tailored to address each error type. These experiments were designed to inform the decision-making process regarding the integration of additional steps to mitigate the occurrence of these errors. Beginning with the 'cannot find symbol' errors, this chapter will delve into the intricacies of error handling and the corresponding experimental findings to elucidate the rationale behind the approach.

Cannot find symbol error handling

In accordance with the findings outlined in the analysis of the Untrained test suite, "cannot find symbol" errors occur when symbols are either incorrectly imported or used within the codebase. To address these errors effectively, it is essential to provide contextual information regarding the package that the symbol may belong to, or the class signature associated with the symbol. As elucidated during the analysis, these errors typically manifest in three distinct categories: class, variable, and method. The class and method categories provide the class name of the symbol, although in different sections of the error information provided by the javac compiler. However, the method category does not explicitly contain the class name within these details. Therefore, our approach to handling each of these categories varies accordingly. For instances categorised under methods, where extracting the class name is unattainable, the focus is on providing the package information if available; otherwise, we resort to presenting the error message alone, as done for general errors. Conversely, for categories encompassing classes and variables, the class name is extracted from the symbol and location attributes of the javac error details, respectively.

To achieve this extraction, a method is employed which utilises regular expressions to match each symbol type within the error details. This method ensures that the class name associated with the symbol is captured accurately, as demonstrated in the pseudo code below:

```
function extract_symbol(string):
    # Define regular expression pattern to match symbol for class category
    pattern = 'symbol:\s+(class|method|variable|static|constructor)\s+([\w.()]+)'
    # Search for pattern in input string
    match = regex_search(pattern, string)
    # If a match is found
    if match is not None:
        # If the first group in the match is None
        if match.group(1) is None:
            # Return 'None' and the symbol extracted from the match
            return ('None', match.group(2))
        else:
            # Define pattern to extract location (class name)
            class_pattern = '\bclass\s+(\w+)'
            # Search for class name in input string
            location_match = regex_search(class_pattern, string)

            # If class name is found
            if location_match is not None:
                # Return the class type, symbol name, and class name
                return (match.group(1), match.group(2), location_match.group(1))
            else:
                # Define pattern to extract location (type name) for variable
                type_pattern = '(?<=\bof type\s)(\w+)'
                # Search for type name in input string
                location_match = regex_search(type_pattern, string)
                # If type name is found
                if location_match is not None:
                    # Return the symbol type, symbol name, and type name
                    return (match.group(1), match.group(2), location_match.group(1))
                # Return the symbol type, symbol name, and None for location
                return (match.group(1), match.group(2), None)
    else:
```

```
        # If no match is found, raise an exception
        raise Exception('This error {string} does not seem to have a symbol')
```

Subsequently, we traverse the source directory of the project to identify all classes defined within it. This task is facilitated by the SourceDirectoryParser class, initialized with the path of the project. The class employs a systematic approach to locate all '.java' files within the directory and utilizes the javalang library to extract class declarations from each file. These classes are then stored and can be queried to ascertain whether a particular class is defined within the project's source code and if so, obtain its corresponding path. The parser class is demonstrated in the following pseudo code:

```
class SourceDirectoryParser:

    // Constructor
    function SourceDirectoryParser(path):
        directory_path = path
        java_files = find_java_files(directory_path)
        classes = parse_java_files(java_files)
        classes = simplify_classes(classes)

    // Find Java files in directory
    function find_java_files(directory):
        java_files = []
        for each file in directory:
            if file ends with '.java':
                add file path to java_files
        return java_files

    // Parse Java files to extract class names and their paths
    function parse_java_files(java_files):
        classes = []
        for each java_file in java_files:
            encoding = detect_encoding(java_file)
            try:
                tree = parse_java_file(java_file)
```

```
            for each node in tree:
                if node is ClassDeclaration:
                    add (class_name, file_path) to classes
        except JavaSyntaxError as e:
            print_syntax_error(java_file, e)
    return classes

// Get path to class file if exists
function get_path(class_name):
    if class_name exists in classes:
        return path corresponding to class_name
    else:
        return None
```

The pseudocode provided above outlines the functionality of the SourceDirectoryParser class, illustrating its role in parsing Java files, extracting class names and paths, and providing access to class paths based on queried class names.

Upon acquiring the path to the symbol, the next objective is to determine the package in which it resides. To accomplish this, a Utilities class is leveraged which performs file path manipulation, enabling the extraction of the package information effectively. Additionally, the aim is to furnish a class signature for the symbol to address cases where errors stem from its incorrect usage. To obtain the class signature, two essential classes are employed: JVMHandler and Reflector.

The Reflector class operates in conjunction with a JVM instance and utilises the Java Reflections API to capture the class signature comprehensively. Upon initialising a JVM instance with a class path listing the required dependencies for file analysis, the Reflector class leverages the Reflections API to extract vital information such as the class definition, constructor signature, and method and field signatures. For each of these components, the Reflector class retrieves details including parameters, modifiers, annotations, and names whenever available, storing these attributes as properties within the class instance. Subsequently, the Reflector object can be queried to obtain the complete class signature or specific constructor, method, or field signatures.

The pseudocode provided below illustrates the functionality of the Reflector class, outlining its operations in accessing class signatures and associated details:

```
class Reflector:

    // Constructor
    function Reflector(jvm_instance):
        jvm = jvm_instance
        reflections = initialize_reflections(jvm)
        class_signatures = extract_class_signatures(reflections)

    // Initialize necessary libraries for Reflection
    function initialize_imports(jvm):
        Class = jpype.JClass('java.lang.Class')
        Method = jpype.JClass('java.lang.reflect.Method')
        Parameter = jpype.JClass('java.lang.reflect.Parameter')
        Constructor = jpype.JClass('java.lang.reflect.Constructor')
        return (Class, Method, Parameter, Constructor)

    // Extract class signatures using Reflections library
    function extract_class_signatures(reflections):
        class_signatures = []
        for each class in reflections.getSubTypesOf(Object):
            class_definition = get_class_definition(class)
            constructor_signature = get_constructor_signature(class)
            method_signatures = get_method_signatures(class)
            field_signatures = get_field_signatures(class)
            add (class_definition, constructor_signature, method_signatures,
            field_signatures) to class_signatures
        return class_signatures

    // functions to retrieve class, constructor, method and field definition
    //...
```

This class serves the purpose of extracting the class signature of the symbol class, thereby enhancing the accuracy of error resolution. Subsequently, armed with both the class signature and package information, our focus shifts towards devising a robust framework to address these

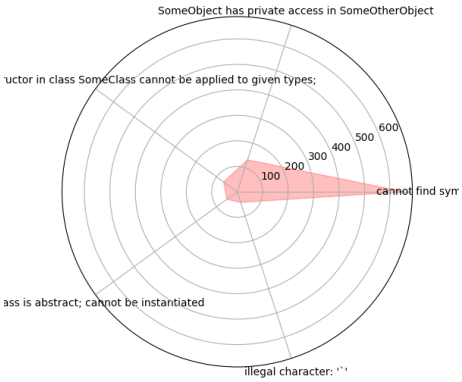
errors systematically. The approach to handling these errors encompasses three distinct scenarios, as previously mentioned. This methodology can be encapsulated through the following pseudocode representation:

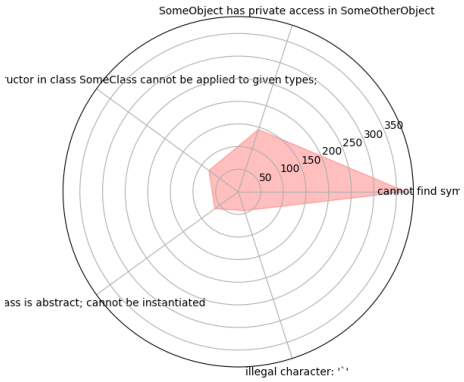
```
function handle_cannot_find_symbol_error(error, parser, reflected):

    // Extract symbol information from error
    symbol_tuple = error.get_additional_property('symbol')
    symbol_type = symbol_tuple[0]
    if symbol_type equals 'variable':
        symbol = symbol_tuple[2]
    else:
                    # class category
        symbol = symbol_tuple[1]

    // Determine symbol file path
    if symbol_type is not equal to 'method':
        symbol_file_path = parser.get_path(symbol)
    else:
        symbol_file_path = None

    // Prepare error messages with contextual information
    error_messages = error.error_details + "\n" + error.details + "\n"

    // If symbol file path is None, return default error messages
    if symbol_file_path is None:
        return error_messages
    else:
        // Get package and reflection of the symbol class
        symbol_package = util.get_package_path(symbol_file_path)
        if not reflected:
            reflecter = Reflecter(class_name=symbol, class_path='',
            package=symbol_package)
            reflection = reflecter.get_reflection_as_string()
```

```
        //If reflection is None, just provide the package
        if reflection is None:
            error_messages += f"\nThis symbol is located in package
            {symbol_package}\n"
        else:
            error_messages += f"\nThis symbol is located in the package
            {symbol_package}. Here's a reflection " \
                                f"of the class signature of the {symbol} class:
                                \n{reflection}\n"
    else:
        error_messages += f"\nThis symbol is located in package
        {symbol_package}\n"
    return error_messages
```

It is imperative to not that the method also takes a reflected parameter which is boolean. This is to ensure that we do not provide more than one class signature for the same class in case the same symbol threw more than one of these errors within a file. The rationale behind the use of this parameter is encapsulated within the error-handling method operating at the file level, which in turn invokes the above method.

Experimentation was conducted with the duplicated untrained test suite, where each "cannot find" error is individually queried to the large language model. This querying process involves providing contextual information such as the package and class signature, if available, utilising the established setup. Remarkably, the number of these errors decreased from 657 to 371, as evidenced by the data depicted in the accompanying box plots.





constructor SomeClassConstructor in class SomeClass cannot be applied to given types

As previously discussed, errors of this nature arise when an object instantiation is attempted with incorrect parameters. Typically, the javac compiler provides information regarding the required parameters, the class name, and a reason for the error. However, due to the stateless nature of the OpenAI API, there is no guarantee that the model, when prompted to rectify the error, will comprehend the original purpose of the object instantiation. Therefore, it becomes essential to augment the contextual information provided to the model. This augmentation involves supplying the class signature of the object and the corresponding package, akin to the approach adopted for handling "cannot find symbol" errors. To implement this augmentation, the following function pseudo-code is employed:

```
function handle_constructor_errors(file_errors, parser):
    // Initialize variables
    reflected_classes = ""
    added_classes = []

    // Iterate through file errors
    for each file_error in file_errors:
        // Get constructor class
        constructor_class = file_error.get_additional_property('class')

        // If a reflection of the constructor class hasnt already been provided
        if constructor_class not in added_classes:
            added_classes.append(constructor_class)

            // Get constructor class location
            constructor_class_location = parser.get_path(constructor_class)

            // If constructor class location is not None
            if constructor_class_location is not None:
                // Get package and reflection information
                constr_package = util.get_package_path(constructor_location)
```

```
                reflecter = Reflecter()
                ref_constructor = reflecter.get_constructors()

                // If constructor signature is found
                if ref_constructor:
                    // Build reflection string
                    reflection = '\nConstructors:\n' + '\n'.join(ref_constructor)
                else:
                    reflection = None

                // If reflection exists, provide package and reflection
                if reflection is None:
                    reflected_classes += f"\nThe class {constructor_class} is located
                else:
                    reflected_classes += f"\nHere's a reflection of the class construc
    return reflected_classes
```

Experimentation within this framework involved reprompting each instance of this error type to the model for resolution. Initially, the total count of such errors stood at 44. Following the reprompting process, all instances were successfully resolved, resulting in a complete eradication of this error type.

package SomePackage does not exist

These errors manifest when the model inappropriately accesses dependencies or incorrectly imports static fields or methods, as discussed earlier in the error analysis stage. The corrective strategy in this setup entails reinforcing adherence to permitted dependencies or adjusting static importations to encompass entire classes rather than specific methods or fields. This is done by providing the permissible dependencies in the context, like it was done in the test generation step.

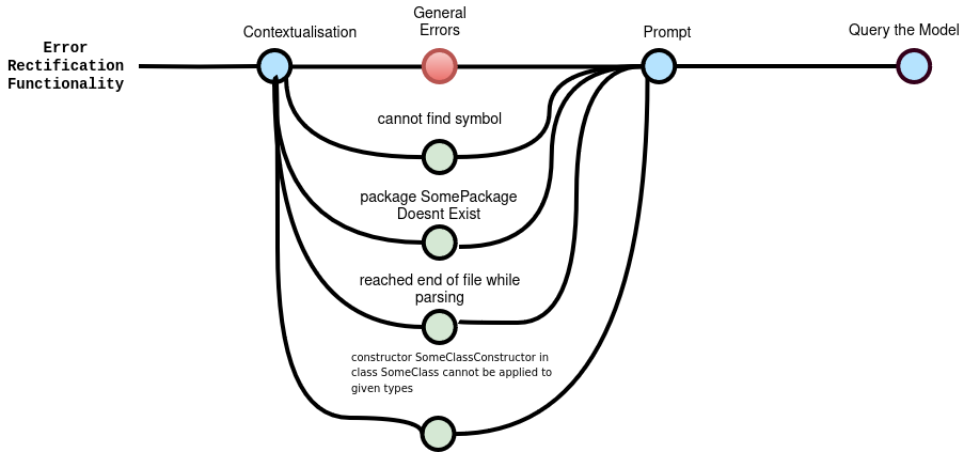
Experimental validation of this approach involved re-prompting each package error to the model alongside pertinent contextual cues. Initially, the total count of package errors stood at 172. Following the implementation adjustments, the error count diminished significantly to 67 accounting for a 61% reduction in total errors if this type.

reached end of file while parsing

The final error type of focus pertains to incomplete implementations within test classes. The rectification involves reprompting the test class alongside the corresponding Class Under Test (CUT), if feasible within the model's token limit. In cases where both entities cannot be accommodated in a single prompt, only the test class is included, with the LLM tasked to complete it. It is important to note that this approach may lead to an upsurge in total errors, as these errors arise in situations where the Java compiler remains unable to parse these test classes for error detection, leaving any arising errors undiscovered.

Experimental verification of this methodology reveals initial instances of 44 such errors within a total of 306 errors. However, upon re-prompting, the overall error count escalated to 710, with only one instance of the targeted error type persisting.

In illustrating the enhanced error rectification process, the current stage encompasses five distinct methodologies for augmenting contextual information within the prompt prior to re-prompting a file and its associated errors during the Error Contextualization and LLM Re-prompting phase. These methodologies are encapsulated within the diagram presented below.



Incorporating this revised setup, error rectification was executed for both the trained and untrained test suites. This process encompassed three stages of re-prompting to the large language model, succeeded by five stages of exclusion of erroneous lines and methods. Subsequently, files containing persistent errors were progressively excluded until no errors remained within each project. Consequently, following this error rectification stage, a total of 940 test files for the trained test suite and 1272 test files for the untrained test suite were retained.

6.9 Results Measurement

Upon completion of error rectification, the evaluation proceeds to assess the test suites based on predefined criteria outlined in earlier sections of this report. As delineated in the design chapter, a structured framework was formulated to gauge the coverage, correctness, and mutation score of each test suite. This section delves into the practical implementation of these frameworks, commencing with coverage analysis.

6.9.1 Coverage

In the pursuit of measuring coverage, Jacoco was selected as the tool of choice to assess the entire dataset. To enact this, Apache Ant tasks were defined to facilitate the instrumentation of compiled tests. Concurrently, modifications were made to the testing targets to enable the Jacoco agent to track coverage as the tests were executed. Additionally, tasks for report generation were incorporated into each project within the dataset, facilitating seamless coverage generation.

While this setup facilitated the generation of coverage reports for the trained, untrained, and human-authored test suites without issue, challenges arose when dealing with EvoSuite-generated tests. A configuration bug was encountered when attempting to integrate EvoSuite with Jacoco which lead to the generation of reports indicating 0

To circumvent this issue, an alternative coverage tool, Cobertura, was employed to measure the coverage of the EvoSuite test suite. This approach necessitated a precaution to be considered when evaluating the results in the Evaluation chapter. Unlike Jacoco, Cobertura does not interfere with EvoSuite's bytecode instrumentation during its coverage measurement process. This distinction arises from the differing instrumentation methods employed by the two tools. Jacoco utilizes on-the-fly instrumentation, which dynamically adds instructions to bytecode at runtime when it is loaded by the JVM. Conversely, Cobertura employs offline instrumentation, which directly adds instructions to bytecode after the compilation of tests. Thus, an Ant task was configured to utilize Cobertura for measuring coverage statistics for each EvoSuite test suite in the evaluation dataset.

6.9.2 Test Correctness

In the design phase, it was stipulated that the assessment of test correctness would be conducted utilizing the Checkstyle tool, which evaluates adherence to the Oracle SUN Java code

convention guide. To implement this, Ant tasks were configured for each test suite, incorporating a Checkstyle agent to analyze the tests. Additionally, an Ant task was assigned to each test suite to generate an XML report containing the findings of Checkstyle's analysis. These XML reports were subsequently consolidated into a singular file for each test suite, amalgamating all reports generated for each project within the dataset. Leveraging these compiled files, graphical representations illustrating the nature of errors were generated using the Matplotlib library in Python.

6.9.3 Mutation coverage

6.10 Wrap up

At the culmination of the implementation chapter, the methodologies defined in the design phase have been diligently executed. Each step was carefully crafted to facilitate the generation and evaluation of all four test suites. However, before proceeding to analyse the results of this evaluation, it is imperative to address legal, professional, social, and ethical considerations within the scope of this report. This will be the focus of the next section.

Chapter 7

Legal, Social, Ethical and Professional Issues

In this section, I address a range of legal, social, ethical, and professional considerations pertinent to the evaluation of a fine-tuned large language model (LLM) for unit testing in Java.

7.1 Copyright Licensing

In conducting the tasks laid out in this report, incorporating code from open-source projects sourced from platforms like GitHub was crucial, hence adherence to copyright and licensing terms was at the forefront of consideration. For the fine tuning task, the projects considered had to come with open source licensing agreements, such as the Apache License 2.0 or GNU General Public License (GPL) to be eligible for selection. In this evaluation, I meticulously adhered to the licensing terms stipulated by the original projects. Selecting projects with compatible licenses ensured that usage was legally compliant. Moreover, this report rigorously followed the attribution guidelines outlined in each license, guaranteeing proper acknowledgment of the original authors and repositories and adherence to the licensing terms throughout this report. This approach upholds legal standards and promotes ethical use of open-source code as per the British Computer Society code of conduct.

Moreover, various open-source libraries and tools, including Jacoco and Checkstyle, were utilized in the development and setup of the evaluation. While these libraries may have differing licenses, they all adhere to open-source principles, thereby permitting free use for the general public. As such, this report strictly adheres to the guidelines set forth by the licenses of these

libraries, ensuring compliance and ethical usage throughout the evaluation process.

7.2 Open Sourcing Evaluation Datasets

Transparency and openness are fundamental tenets of scientific research, particularly in the field of machine learning and artificial intelligence. To uphold these principles, all evaluation datasets as well as the design setup are openly accessible to the public. By sharing these datasets, which encompass both training and evaluation data, along with the code utilised during the evaluation process, the aim is to facilitate reproducibility, enable peer review, and foster collaboration within the research community. These datasets will be made available through designated repositories or platforms, ensuring accessibility while safeguarding data privacy and anonymising sensitive information such as API keys. By making these datasets open source, I adhere to ethical standards in research and contribute to the advancement of knowledge in the field.

7.3 Ethical Use of Large Language Models

The deployment of large language models (LLMs) for diverse tasks raises profound ethical considerations. In the context of unit testing in Java, it is imperative to address potential ethical implications, including biases in model predictions, unintended consequences of model outputs, and the ethical responsibility of researchers. i recognise the significance of considering issues related to data privacy, algorithmic transparency, and the societal impact of LLM-generated code on software development practices. Adhering to ethical guidelines, such as fairness, accountability, and transparency, is paramount throughout the research process involving LLMs. I am committed to upholding these principles and contributing to the responsible use of AI technologies in software engineering practices, thereby promoting ethical conduct and social responsibility in research endeavours.

Chapter 8

Results/Evaluation

8.1 Overview

This chapter presents the results of the evaluation conducted to assess the effectiveness of the test suites generated. The evaluation focused on three key criteria: code coverage, mutation score, and test correctness. Four distinct test suites were examined: human-written tests, tests generated by EvoSuite, tests generated by a fine-tuned LLM model, and tests generated by an ordinary LLM model. Each test suite was subjected to rigorous analysis to evaluate its ability to adequately test software applications and detect faults.

The assessment of code coverage provides insights into the proportion of code exercised by the test suites, shedding light on their thoroughness in exploring different paths and functionalities within the software under test. Mutation score analysis, on the other hand, offers a deeper evaluation of the test suites' ability to detect faults by measuring their effectiveness in killing mutant code variations. Additionally, test correctness evaluation examines the readability and maintainability of the test suites by evaluating their adherence to well known code style conventions.

By evaluating these test suites across multiple dimensions, the overarching aim is to provide a comprehensive understanding of their strengths, weaknesses, and overall effectiveness. The findings presented in this chapter offer valuable insights into the capabilities of large language models in automated test generation and their potential impact on software testing practices.

Let's move on to the detailed presentation of the evaluation results, starting with the assessment of code coverage

8.2 Coverage Assessment

Coverage analysis plays a crucial role in evaluating the effectiveness of test suites by providing insights into the extent to which the codebase is exercised during testing. It measures the proportion of code lines and branches that are executed by the test suite, indicating the thoroughness of test coverage in exploring different paths and functionalities within the software under test. A high coverage percentage suggests that the test suite is comprehensive and likely to detect more defects, while a low coverage percentage may indicate gaps in the testing strategy, potentially leaving critical areas of the code untested.

In terms of coverage, the results obtained from the evaluation of the test suites are as follows:

1. Untrained Test Suite:

   Line Coverage: 72.69

   Branch Coverage: 59.125

2. Trained Test Suite:

   Line Coverage: 59.125

   Branch Coverage: 34

3. Evosuite Test Suite*:

   Line Coverage: 61.8

   Branch Coverage: 28.4

4. Human Authored Test Suite:

   Line Coverage: 63.60

   Branch Coverage: 52.18

Note: The Evosuite test suite was measured using a different tool compared to the other test suites. This difference in measurement tools should be taken into consideration when interpreting the results.

The coverage analysis reveals significant variations in the coverage percentages across different test suites. The untrained test suite exhibits the highest line and branch coverage percentages, suggesting thorough exploration of the codebase. However, the trained test suite and Evosuite test suite demonstrate lower coverage percentages, indicating potential gaps in test coverage. The human-authored test suite falls between the untrained and trained test suites in

terms of coverage percentages, showcasing a balanced approach to test coverage. These findings underscore the importance of comprehensive coverage analysis in evaluating the effectiveness of test suites and identifying areas for improvement in automated test generation approaches.

8.3 Mutation Score Assessment

Mutation testing is a powerful technique used to evaluate the effectiveness of test suites by introducing small changes (mutations) to the codebase and assessing whether the tests are able to detect these mutations. A high mutation score indicates that the test suite is capable of identifying and capturing faults in the code, whereas a low mutation score suggests that the test suite may lack the ability to effectively detect defects within the codebase.

The mutation score results obtained from the evaluation of the test suites are as follows:

1. Untrained Test Suite:

   Mutation Coverage: 21

2. Trained Test Suite:

   Mutation Coverage: 16

3. Evosuite Test Suite:

   Mutation Coverage: 41

4. Human Authored Test Suite:

   Mutation Coverage: 36

The mutation score analysis reveals notable disparities in the effectiveness of the test suites in detecting mutations. The Evosuite test suite achieves the highest mutation coverage, indicating its ability to detect a wide range of faults in the codebase. Conversely, the trained test suite demonstrates lower mutation coverage, suggesting potential weaknesses in the test suite's ability to detect mutations introduced into the code. The human-authored test suite falls between the Evosuite and trained test suites, showcasing a moderate ability to identify faults through mutation testing. These findings underscore the importance of mutation testing in assessing the robustness of test suites and identifying areas for improvement in automated test generation approaches.

8.4 Test Correctness Analysis

Test correctness refers to the extent to which test suites adhere to code style conventions, such as the Oracle SUN Java Code Style Guide. Evaluating test correctness is crucial for ensuring the readability and maintainability of test suites, which are essential for effective software development and maintenance. Adhering to code style conventions promotes consistency and clarity in codebases, making it easier for developers to understand and modify the code. Consistently formatted test suites enhance code readability, facilitate code reviews, and contribute to overall software quality and maintainability.

The assessment of test correctness based on adherence to the Oracle SUN Java Code Style Guide yielded the following results:

1. Untrained Test Suite:

   Total Violations: 72.69

   Average Violations per project: 276.2 violations

2. Trained Test Suite:

   Total Violations: 59.125

   Average Violations per project: 202.9 violations

3. Evosuite Test Suite*:

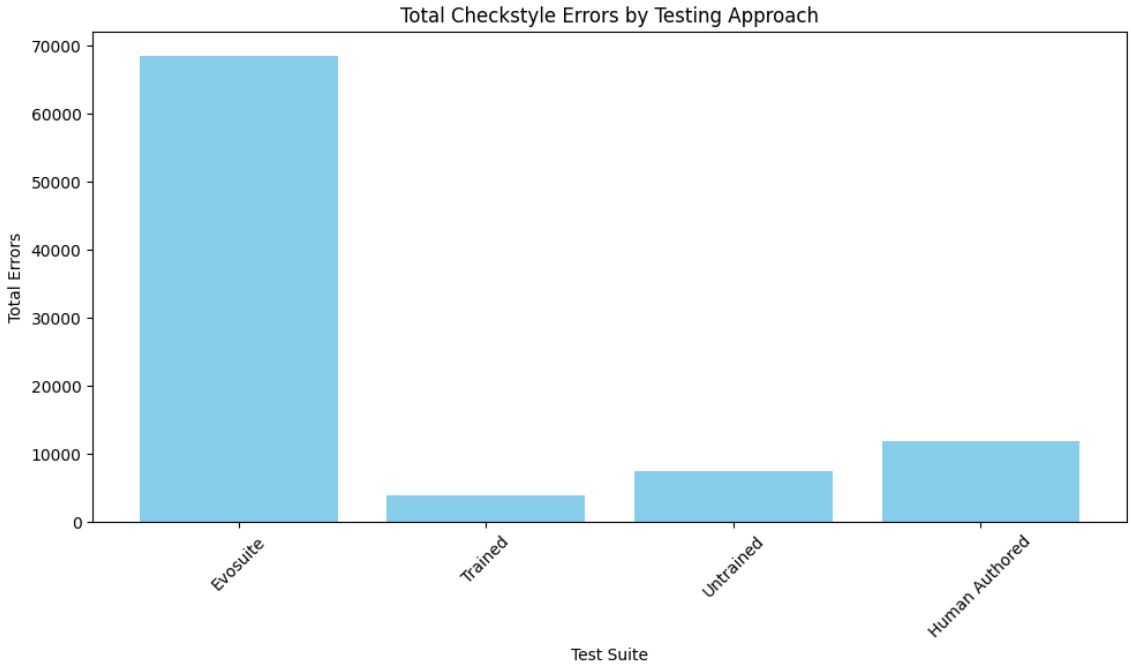
   Total Violations: 61.8

   Average Violations per project: 3609 violations

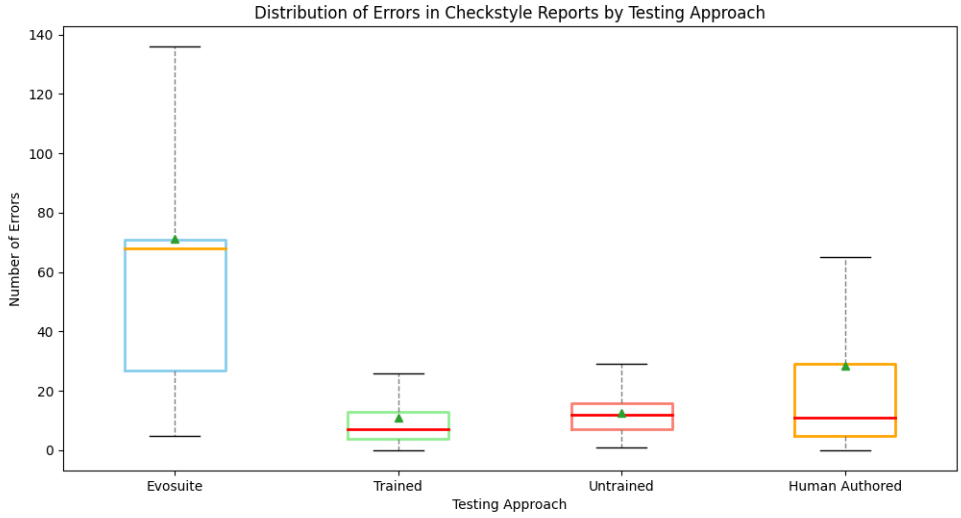
4. Human Authored Test Suite:

   Total Violations: 63.60

   Average Violations per project: 629 violations

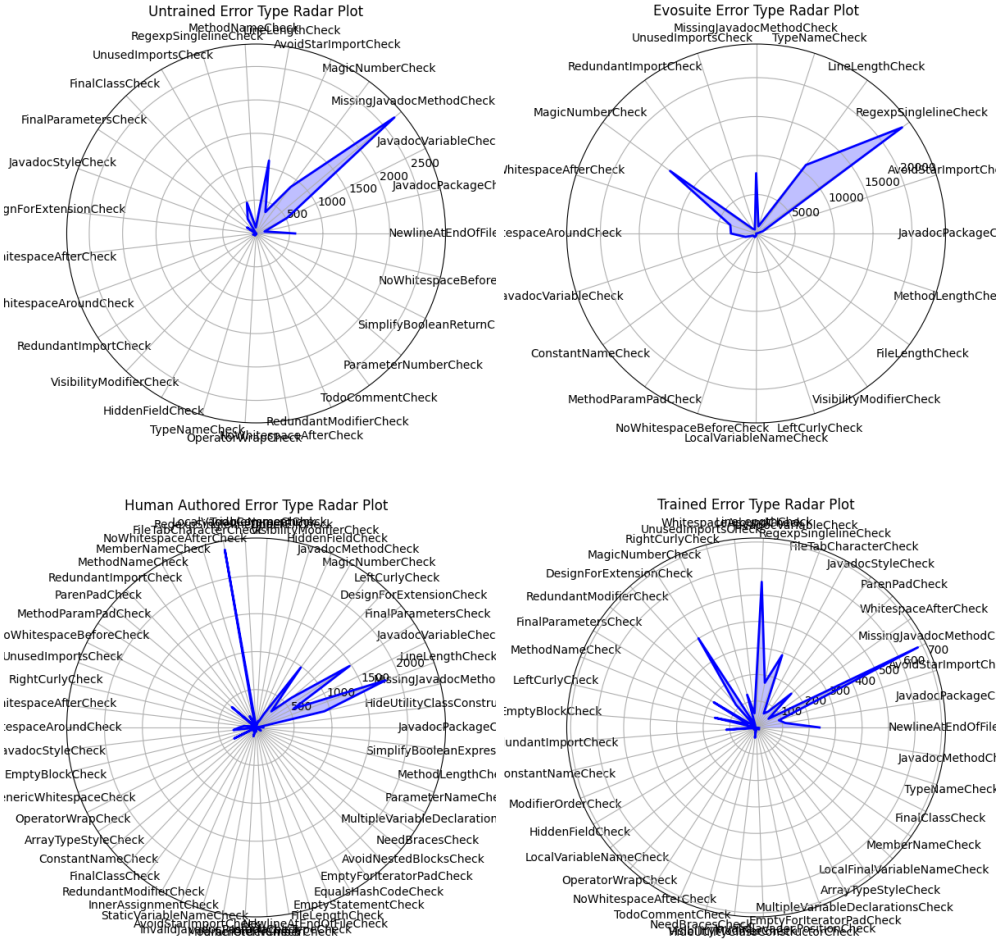


To further showcase the distribution of these violations per file the following box plot matrix showcases the spread of these violations per project for each of the test suites.



The analysis of test correctness reveals significant differences in adherence to code style conventions among the test suites. The Evosuite test suite exhibits the highest total violations and average violations per project, indicating a greater divergence from code style conventions. Conversely, the trained test suite demonstrates lower total violations and average violations per project, suggesting better adherence to code style guidelines. These findings underscore the importance of incorporating code style conventions into automated test generation processes to improve test readability and maintainability.

It is imparative to also explore the spread of the types of violations for each of these test methodologies. To achieve this the following radar plots demonstrate the main violations in each of the test suites:



8.5 Synopsis

The evaluation reveals varying strengths and weaknesses across the test suites. While the untrained test suite demonstrated higher coverage metrics, it exhibited poorer mutation scores and code style adherence. In contrast, the Evosuite test suite displayed superior mutation scores but suffered from higher code style violations. The trained test suite showed improvements in code style adherence but lagged behind in coverage and mutation score metrics. Overall, the human-authored test suite performed competitively across all criteria, emphasizing the value of manual test creation in ensuring comprehensive and high-quality test suites.

Chapter 9

Conclusion and Future Work

The project's conclusions should list the key things that have been learnt as a consequence of engaging in your project work. For example, "The use of overloading in C++ provides a very elegant mechanism for transparent parallelisation of sequential programs", or "The overheads of linear-time n-body algorithms makes them computationally less efficient than $O(n \log n)$ algorithms for systems with less than 100000 particles". Avoid tedious personal reflections like "I learned a lot about C++ programming...", or "Simulating colliding galaxies can be real fun...". It is common to finish the report by listing ways in which the project can be taken further. This might, for example, be a plan for turning a piece of software or hardware into a marketable product, or a set of ideas for possibly turning your project into an MPhil or PhD.

References

[1] Apache license.

[2] Gnu general public license.

[3] Azat Abdullin, Marat Akhin, and Mikhail Belyaev. Kex at the 2022 sbst tool competition. In *Proceedings of the 15th Workshop on Search-Based Software Testing*, pages 35–36, 2022.

[4] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. A3test: Assertion-augmented automated test case generation. *arXiv preprint arXiv:2302.10352*, 2023.

[5] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272. IEEE, 2017.

[6] Ellen Arteca, Sebastian Harner, Michael Pradel, and Frank Tip. Nessie: automatically testing javascript apis with asynchronous callbacks. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1494–1505, 2022.

[7] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*, 2022.

[8] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[9] CheckStyle. Checkstyle.

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[11] Pouria Derakhshanfar and Xavier Devroey. Basic block coverage for unit test generation at the sbst 2022 tool competition. In *Proceedings of the 15th Workshop on Search-Based Software Testing*, pages 37–38, 2022.

[12] Filomena Ferrucci, Mark Harman, and Federica Sarro. Search-based software project management. *Software project management in a changing world*, pages 373–399, 2014.

[13] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[14] Gordon Fraser and Andrea Arcuri. Sound empirical evidence in software testing. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 178–188. IEEE, 2012.

[15] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2012.

[16] Gordon Fraser and Andrea Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):1–42, 2014.

[17] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, pages 206–215, 2008.

[18] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[19] Google. Google java style guide.

[20] G. Grano, F. Palomba, D. D. Nucci, A. D. Lucia, and H. C. Gall. Scented since the beginning: on the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software*, 156:312–327, 2019.

[21] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):1–61, 2012.

[22] HuggingFace. Starcoder: A state-of-the-art llm for code.

[23] Google Inc. Guava.

[24] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.

[25] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *International conference on software engineering (ICSE)*, 2023.

[26] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

[27] Adam Lipowski and Dorota Lipowska. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications*, 391(6):2193–2196, 2012.

[28] Phil McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[29] Microsoft. Microsoft methods2test.

[30] Brad L Miller, David E Goldberg, et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.

[31] M. Motwani. High-quality automated program repair. 2021.

[32] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. A conversational paradigm for program synthesis, 2022.

[33] Mark O'Keeffe and Mel Ó Cinnéide. Search-based software maintenance. In *Conference on software maintenance and reengineering (CSMR'06)*, pages 10–pp. IEEE, 2006.

[34] OpenAI. Chatgpt (2021) openai.

[35] OpenAI. Chatgpt (2021) openai.

[36] OpenAI. Openai guide to fine tuning documentation.

[37] Oracle. Sun java style guide.

[38] Jacoco Org. Jacoco: Java code coverage library.

[39] OWASP. Owasp defectdojo.

[40] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

[41] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84. IEEE, 2007.

[42] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*, pages 1–10. IEEE, 2015.

[43] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2017.

[44] Pitest.Org. Pitest mutation testing.

[45] Spring projects. springboot.

[46] ReactiveX. Rxjava.

[47] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *arXiv preprint arXiv:2302.06527*, 2023.

[48] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[49] S. Shamshiri, J. M. Rojas, J. P. Galeotti, and N. Walkinshaw. How do automatically generated unit tests influence software maintenance? *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018.

[50] Mohammed Latif Siddiq, Joanna Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. Exploring the effectiveness of large language models in generating unit tests. *arXiv preprint arXiv:2305.00418*, 2023.

[51] springprojects. Spring-framework.

[52] square. Retrofit.

[53] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. Chatgpt vs SBST: A comparative assessment of unit test suite generation. *CoRR*, abs/2307.00588, 2023.

[54] TheAlgorithms. Thealgorithmsjava.

[55] Paolo Tonella. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes*, 29(4):119–128, 2004.

[56] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*, 2020.

[57] Shengcheng Yu, Chunrong Fang, Yuchen Ling, Chentian Wu, and Zhenyu Chen. Llm for test script generation and migration: Challenges, capabilities, and opportunities. *arXiv preprint arXiv:2309.13574*, 2023.

[58] Yuanyuan Zhang, Anthony Finkelstein, and Mark Harman. Search based requirements optimisation: Existing work and challenges. In *Requirements Engineering: Foundation for Software Quality: 14th International Working Conference, REFSQ 2008 Montpellier, France, June 16-17, 2008 Proceedings 14*, pages 88–94. Springer, 2008.

[59] Zhichao Zhou, Yuming Zhou, Chunrong Fang, Zhenyu Chen, and Yutian Tang. Selectively combining multiple coverage goals in search-based unit test generation. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

Appendix A

Extra Information

A.1 Tables, proofs, graphs, test cases, ...

The appendices contain information that is peripheral to the main body of the report. Information typically included in the Appendix are things like tables, proofs, graphs, test cases or any other material that would break up the theme of the text if it appeared in the body of the report. It is necessary to include your source code listings in an appendix that is separate from the body of your written report (see the information on Program Listings below).

Appendix B

User Guide

B.1 Instructions

You must provide an adequate user guide for your software. The guide should provide easily understood instructions on how to use your software. A particularly useful approach is to treat the user guide as a walk-through of a typical session, or set of sessions, which collectively display all of the features of your package. Technical details of how the package works are rarely required. Keep the guide concise and simple. The extensive use of diagrams, illustrating the package in action, can often be particularly helpful. The user guide is sometimes included as a chapter in the main body of the report, but is often better included in an appendix to the main report.

Appendix C

Source Code

C.1 Instructions

Complete source code listings must be submitted as an appendix to the report. The project source codes are usually spread out over several files/units. You should try to help the reader to navigate through your source code by providing a "table of contents" (titles of these files/units and one line descriptions). The first page of the program listings folder must contain the following statement certifying the work as your own: "I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary". Your (typed) signature and the date should follow this statement.

All work on programs must stop once the code is submitted to KEATS. You are required to keep safely several copies of this version of the program and you must use one of these copies in the project examination. Your examiners may ask to see the last-modified dates of your program files, and may ask you to demonstrate that the program files you use in the project examination are identical to the program files you have uploaded to KEATS. Any attempt to demonstrate code that is not included in your submitted source listings is an attempt to cheat; any such attempt will be reported to the KCL Misconduct Committee.

You may find it easier to firstly generate a PDF of your source code using a text editor and then merge it to the end of your report. There are many free tools available that allow you to merge PDF files.