Automated Discovery of Internet-Reported Software Bugs and AI-Driven Test Case Generation

by

Subrata Kumar Bhowmik
ID: CSE2203027119

Biplob Chakma
ID: CSE2203027120

Md Ebtesham Hossain
ID: CSE2203027054

Shumen Baral
ID: CSE2203027150

Supervised by
Tarunnyamoye Kundu

Submitted in partial fulfillment of the requirements for the degree of
Bachelor of Science in Computer Science and Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SONARGAON UNIVERSITY (SU)
May 2026

Automated Discovery of Internet-Reported Software Bugs and AI-Driven Test Case Generation

by

Subrata Kumar Bhowmik
ID: CSE2203027119

Biplob Chakma
ID: CSE2203027120

Md Ebtesham Hossain
ID: CSE2203027054

Shumen Baral
ID: CSE2203027150

Supervised by
Tarunnyamoye Kundu

Submitted in partial fulfillment of the requirements for the degree of
Bachelor of Science in Computer Science and Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SONARGAON UNIVERSITY (SU)
May 2026

APPROVAL

The thesis titled "**Automated Discovery of Internet-Reported Software Bugs and AI-Driven Test Case Generation.**" submitted by Subrata Kumar Bhowmik (CSE2203027119), Biplob Chakma (CSE2203027120), Md Ebtesum Hossain(CSE2203027054) and Shumen Baral (CSE2203027150) to the Department of Computer Science and Engineering, Sonargaon University (SU), has been accepted as satisfactory for the partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science and Engineering and approved as to its style and contents.

Board of Examiners

Tarunnyamoye Kundu — **Supervisor**
Lecturer,
Department of Computer Science and Engineering
Sonargaon University (SU)

(Examiner Name and Signature) — **Examiner 1**
Department of Computer Science and Engineering
Sonargaon University (SU)

(Examiner Name and Signature) — **Examiner 2**
Department of Computer Science and Engineering
Sonargaon University (SU)

(Examiner Name and Signature) — **Examiner 3**
Department of Computer Science and Engineering
Sonargaon University (SU)

DECLARATION

We, hereby, declare that the work presented in this report is the outcome of the investigation performed by us under the supervision of **Tarunnyamoye Kundu, Lecturer,** Department of Computer Science and Engineering, Sonargaon University, Dhaka, Bangladesh. We reaffirm that no part of this thesis has been or is being submitted elsewhere for the award of any degree or diploma.

Countersigned

(Tarunnyamoye Kundu)
Supervisor

Signature

Subrata Kumar Bhowmik
ID: CSE2203027119

Biplob Chakma
ID: CSE2203027120

Md Ebtesham Hossain
ID: CSE2203027054

Shumen Baral
ID: CSE2203027150

ABSTRACT

Software bugs reported on platforms like GitHub, Google Play, and Stack Overflow represent a vast, untapped source of real-world failure data. While automated test case generation has advanced with large language models, existing approaches operate only on source code or specifications and ignore user-reported defects. This thesis presents an end-to-end framework that automatically discovers, structures, and converts internet-reported bugs into executable test cases. The framework comprises of four integrated phases: multi-source bug collection from GitHub Issues and Play Store reviews using label-based filtering; NLP analysis using TF-IDF and a Naïve Bayes classifier with rule-based extraction; test case generation via LLMs (GPT-4, GPT-3.5-turbo, LLaMA-2 7B) using engineered prompts with an abstention mechanism; and export to standard formats. A key feature is principled refusal to generate low-quality test cases, prioritizing precision over recall. The Framework is evaluated on 120 annotated bug reports from three open-source repositories and five mobile apps, results show collection precision of 0.86, classification F1 of 0.83, and GPT-4 composite test-case quality of 0.93. Generation quality correlates with input completeness, validating the abstention design. This research establishes internet-reported bugs as a viable input for AI-driven testing. The modular open-source implementation supports independent reuse of each pipeline phase.

ACKNOWLEDGMENT

At the very beginning, we would like to express our deepest gratitude to the Almighty God for giving us the ability and the strength to finish the task successfully within the schedule time.

We are auspicious that we had the kind association as well as supervision of **Tarunnyamoye Kundu**, Lecturer, Department of Computer Science and Engineering, Sonargaon University whose hearted and valuable support with best concern and direction acted as necessary recourse to carry out our thesis.

We would like to convey our special gratitude to **Bulbul Ahmed,** for his kind concern and precious suggestions.

We are also thankful to all our teachers during our whole education, for exposing us to the beauty of learning.

Finally, our deepest gratitude and love to our parents for their support, encouragement, and endless love.

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
AUC-ROC	Area Under ROC Curve
BERT	Bidirectional Encoder Representations from Transformers
CAT-ML	Code And Tests Language Model
CD	Continuous Development
CI	Continuous Integration
CoT	Chain-of-Thought
FN	False Negative
FP	False Positive
GA	Genetic Algorithm
GPT	Generative Pre-trained Transformer
GVR	Generation-Validation Repair
IR	Information Retrieval
LLaMA	Large Language Model Meta AI
ML	Machine Learning
NER	Named Entity Recognition
NLP	Natural Language Processing
OS	Operating System
SBST	Search-based Software Testing
SDLC	Software Development Life Cycle
SQA	Software Quality Assurance
TC	Test Case
TF	Term Frequency
TF-IDF	Term Frequency Inverse Document Frequency
TN	True Negative
TPR	True Positive Rate
TP	True Positive
ToT	Tree-of-Thought
UI	User Interface
UML	Unified Modeling Language

TABLE OF CONTENTS

Title Automated Discovery of Internet-Reported Software Bugs and AI-Driven Test Case Generation Page No.

DECLARATION ……..…….. iii

ABSTRACT ……...……. iv

ACKNOWLEDGEMENT …….…… v

LIST OF ABBREVIATION …….……. vi

CHAPTER 1 1 – 9

INTRODUCTION

1.1 Introduction …….....……..... 1

1.2 Software Bugs and Test Case ..……...……...……..... 1 - 2

1.3 Artificial Intelligence (AI) …….. 2

1.4 AI-Driven Test Case …….……....…...…..... 2 - 3

1.5 Literature Review …..…….....……..... 3

1.5.1 Background…….. 3 - 4

1.5.2 Methods of Genetic Algorithm and Information Retrieval …. 4

1.5.3 AI-Powered Frameworks for Test Generation …… 4 - 5

1.5.4 LLM-Based Test Case Generation …….. 5 - 8

1.5.5 Research Gap …….. 8

1.6 Objectives …….…….…….……. 8 - 9

1.7 Scopes and Limitations …… 9

CHAPTER 2 10 - 18

METHODOLOGY

2.1 Introduction …….…..... 10

2.2 Bug Collection Methodology…….…… 10

2.2.1 Data Source Selection …….. 10- 11

2.2.2 GitHub Issue Collection Protocol …… 12

2.2.3 Google Play Store Collection Protocol …… 12 - 13

2.3 NLP Analysis Pipeline Methodology …….……. 13

2.3.1 Bug Classification Approach …..……. 13 – 14

2.3.2 Information Extraction Methodology …….. 14 - 15
2.3.3 Normalization and Deduplication …… 15 - 16
2.4 Test Case Generation Methodology ……. 16
2.4.1 Large Language Model Selection and Configuration …….. 16
2.4.2 Prompt Engineering Strategy ……... 16
2.4.3 Test Case Generation Workflow ……. 16 - 17
2.4.4 Multi-Mode Generation …….. 17
2.5 Validation Methodology …….. 17
2.5.1 Evaluation Datasets ……. 17 - 18
2.5.2 Evaluation Metric …… 18
2.6 Implementation Technologies …….. 18

CHAPTER 3 19 – 25
SYSTEM ARCHITECTURE AND DESIGN
3.1 Introduction …….. 19
3.2 High Level Architecture……..... 19 - 20
3.3 Bug Collection Module…….…… 21
3.2.1 GitHub Issues Collector…….…….…..... 21
3.2.2 Google Play Store Collector…….……... 21 - 22
3.4 NLP Analysis Pipeline…….. 22
3.4.1 Bug Classifier ……. 22
3.4.2 Information Extraction Module …….. 22 - 23
3.4.3 Normalization Module …… 23
3.5 Test Case Generation Engine…… 23 - 24
3.5.1 LLM Client ……. 24
3.5.2 Prompt Construction Module …….. 24
3.5.3 Test Case Generator …… 24 - 25
3.6 Export and Reporting Module…….. 25

CHAPTER 4 26 – 41
IMPLEMENTATION
4.1 Introduction…….…… 26
4.2 Data Flow Overview……. 26

4.2.1 Collection Phase……..…… 26 - 27
4.2.2 Analysis Phase……. 27
4.2.3 Generation Phase…….. 27
4.2.4 Export Phase …… 27
4.3 Bug Collection Implementation …… 28
4.3.1 GitHub Issue Collector …… 28 - 29
4.3.2 Bug Detection Pattern …….. 30
4.3.3 Base Collector Class ……. 30 - 31
4.4 Analyzer Module Implementation …… 31
4.4.1 NLP Processor Component …….. 31 - 32
4.4.2 Bug Classifier Component ……... 32 – 33
4.4.3 Bug Categories ……. 33
4.4.4 Severity Levels ……. 33
4.5 Generator Module Implementation …….. 34
4.5.1 Test Case Generation Process ……. 34 - 35
4.5.2 Test Type Generated ……... 35 – 36
4.6 Web Module Implementation …….. 36
4.6.1 Flask Web Application …… 36
4.6.2 Software and Hardware Requirement ……. 37
4.7 Export and Reporting …… 37 - 38
4.7.1 Excel Output Schema …….. 38 – 39
4.7.2 End-to-End Pipeline Summary …… 39
4.8 Sample Execution ……. 39 - 41

CHAPTER 5 42 – 47
EXPERIMENTAL RESULTS
5.1 Introduction ……..……. 42
5.2 Implementation Summary …… 42
5.3 Collection Accuracy …… 43
5.3.1 Collection Performance Overview ……. 43
5.3.2 GitHub Collection Results (Dataset A) …….. 43
5.3.3 Play Store Collection Results (Dataset B) ……. 43
5.4 Analysis Effectiveness …… 44

5.4.1 Classification Performance …… 44
5.4.2 Classification by Bug Category ……..…… 44
5.4.3 Severity Classification Accuracy ……... 44
5.4.4 Information Extraction Performance ……. 45
5.5 Test Case Generation Quality ……. 45
5.5.1 Generation Success by LLM Backend …… 45
5.5.2 Generation Quality by Bug Severity ……..…….. 46
5.5.3 Generation Quality by Bug Category ……. 46
5.5.4 Test Type Distribution …… 46
5.6 Comparison with Relevant Methods …….. 47

CHAPTER 6 48 - 49
CONCLUSION AND FUTURE WORKS
6.1 Conclusion ……..….. 48 - 49
6.2 Future Works …….…… 49
6.2.1 Enhanced Collection Infrastructure ……. 49 – 50
6.2.2 Advanced NLP and Classification …….. 50
6.2.3 Test Case Execution and Validation ……... 50
6.2.4 Bug Report Quality Assessment ……. 50
6.2.5 Multi-Modal Bug Reports ……... 50
6.2.6 Cross-Language and Cross-Platform Support ……. 51
6.2.7 Integration with Bug Prediction and Prioritization ……. 51
6.2.8 Explainable Test Case Generation …….. 51

REFERENCES ……..…….. 52 – 54

LIST OF TABLES

Table No.	Title	Page No.
Table 2.1	Technologies used in implementation …….	18
Table 3.1	The four-stage pipeline of the framework ……...	19
Table 3.2	Roles of GitHub Issues Collector ……	21
Table 3.3	Roles of Google Play Store Collector …….	22
Table 3.4	Rule-based NLP patterns to extract structured field from the raw text …….	23
Table 4.1	Supported Bug Collection Sources ……	28
Table 4.2	Bug Detection Patterns and Examples ……..	30
Table 4.3	Collected Bug Report Fields ……..	31
Table 4.4	Bug Classification Categories ……	33
Table 4.5	Bug Severity Levels ……	33
Table 4.6	Generated Test Case Types …….	35
Table 4.7	Factors Influencing Test Case Count ……..	35
Table 4.8	Key REST API Endpoints ……..	36
Table 4.9	Software Dependencies ……..	37
Table 4.10	External Service Dependencies ……..	37
Table 4.11	Hardware Requirements …….	37
Table 5.1	Collection Accuracy Metrics ……..	43
Table 5.2	GitHub Collection Breakdown by Repository ……	43
Table 5.3	Play Store Collection by Keyword Category ……..	43
Table 5.4	Classification Metrics by Source ……	44
Table 5.5	Classification Performance by Bug Category …….	44
Table 5.6	verity Classification Results ……	44
Table 5.7	Extraction Success Rate by Field ……	45
Table 5.8	Generation Quality Metrics by LLM Backend ……	45

Table 5.9 Generation Quality by Severity Level …… 46
Table 5.10 Generation Quality by Bug Category ……. 46
Table 5.11 Generated Test Type Distribution by Bug Category ……. 46
Table 5.12 Overall Generation Quality Comparison …… 47

LIST OF FIGURES

Figure No.	Title	Page No.
Fig.2.1	Data Flow Diagram ……	11
Fig.2.2	Overview of information extraction methodology ……	15
Fig.3.1	High-Level Architecture of the AI-Driven TC ……..	20
Fig.5.1	Field Extraction Comparison …….	45

CHAPTER 1

INTRODUCTION

1.1 Introduction

Modern society relies on software systems, which underpin critical infrastructure, global commerce and daily communication. As the size and complexity of these systems increase, the challenge of assuring their reliability grows as well. Software bugs or defects that lead to unexpected behavior or system failure - are a perennial and expensive problem. These bugs are increasingly reported by users through internet platforms such as Google Play Store, GitHub issue trackers, Stack Overflow discussions, and dedicated bug repositories, resulting in large, unstructured datasets of real-world software failures. At the same time, software testing, which is the main line of defense against such defects, is being transformed by using Artificial Intelligence (AI) to automate and optimize the generation of test cases.

Traditional software testing methodologies have two major limitations. First, manual test case design is laborious, time-consuming and usually cannot provide sufficient coverage for complex large-scale software systems. To solve this problem, researchers have created automated test case generation techniques using methods such as fuzzy logic, fault propagation path coverage, and coverage metrics [1] [2]. But these automated approaches have a second limitation: they usually work from the code structure or from specifications laid down beforehand, not from real-world usage data. As a result, the specific scenario that leads to failures in real user environments may even go unnoticed by automated techniques. The wealth of information in software bugs reported on the Internet - echoing real user experiences and edge cases, remains largely unexploited for guiding test case generation. This gap creates an opportunity for AI to automate the test cases, with machine learning and natural language processing techniques potentially helping to bridge the gap between structured code analysis and unstructured real-world failure data [3] [4].

Both areas of automated test case generation and software bug analytics have been studied extensively, but there has been little research investigating the intersection of these fields. Existing studies have not sufficiently investigated how the unstructured and ecologically valid data from Internet-reported bugs can be systematically exploited by AI techniques to inform and improve the test case generation process. A framework that connects real world user-reported failures to intelligent, automated testing could dramatically improve software reliability by directing testing efforts towards the most relevant and impactful scenarios [5] [6] [7].

1.2 Software Bug and Test Case

Software Bug

A software bug is an error, flaw or fault in a computer program or software that causes it to produce an unintended result or to crash or behave in unintended ways. Bugs are caused by human errors in

design or coding, and they can range from annoying visual glitches to catastrophic system failures. These are identified through testing and resolved through debugging.

That is, a bug is a flaw in the software/system that may cause the system or components to fail in performing their required functions. Incorrect data description, statements, input data, design, etc. For example.

The most common types of bugs are syntax errors, which happen when the code breaks the rules of the programming language; runtime errors, which occur while the program is running; and logic errors, which occur when the program doesn't run as expected due to faulty reasoning.

One of the most important aspects of the software development is to find and fix the defects. This process often involves extensive testing, debugging tools and cooperation of developers.

Test Case

A test case is a set of conditions or steps used to verify the functionality of a particular feature or component of a software application. It is one of the important phase of software testing process in which the software behaves as expected in different scenarios.

Best practices of test case, includes ensuring test cases are clear, concise, and easy to understand. It covers all possible scenarios, including edge cases. Regularly update the test cases to reflect changes in the software.

1.3 Artificial Intelligence (AI)

Artificial intelligence (AI) is the ability of machines to ingest information, learn from it, and act intelligently toward a specific goal.

AI is the use of data, algorithms and computing power to imitate or augment human thinking and problem solving. Algorithms are a part of the structure of the AI. Simple algorithms are used for simple applications, while more complex ones help to frame strong artificial intelligence.

Machine learning (ML) is a subset of AI, a notion that computer programs can learn from and adapt to new data without human help. AI allows machines to learn from experience, adapt to new inputs and carry out tasks that were once only possible for humans to do. The vast majority of the AI that we hear about today - from computers that can play chess to self-driving cars - rely heavily on deep learning and natural language processing. The technologies allow computers to be trained to perform certain tasks by feeding them massive amounts of data and letting them find patterns in that data. [8] [9].

1.4 AI-Driven Test Case (TC)

AI driven test case means the Test Cases generated by using AI models and machine learning techniques for software applications. This approach enables automatic test case generation and enhances the efficiency, coverage and effectiveness of the testing process

by reducing manual effort and identifying potential issues in a more comprehensive manner.

1.5 Literature Review

1.5.1 Background

Bugs reported in online platforms, issue trackers, mailing lists, user forums and version control systems are unique challenges for software bug automation. The bugs reported on the Internet are described in natural language, not as formally specified defects. They can be ambiguous, incomplete or context-dependent. To deal with this, Hooda and Chhillar (2018) mentioned that for large software projects, bug report systems such as Bugzilla receive hundreds to thousands of reports, making manual triage and test case derivation impossible. Therefore, effective retrieval and localization of relevant code artifacts from natural language bug reports has been a critical research priority. Among the earliest automated approaches developed are information retrieval (IR) based methods [3].

The authors used common IR techniques (e.g., vector space models and similarity functions based on BM25) to map natural language bug descriptions to relevant source files. These approaches were computationally tractable, but they failed to capture the semantic nuance and the structure of the code, so richer representations and learning based methods were adopted.

Test case generation is known to be one of the most intellectually demanding and resource consuming tasks in software testing, often accounting for more than 50% of total development costs [3]. Existing automated techniques include search-based software testing (SBST), constraint-based generation, model-based testing and random testing. Frameworks like EvoSuite (Java) and Pynguin (Python) represent classical automation, achieving reasonable code coverage by optimization-guided exploration of the input space.

However, as Yixuan Chen (2024) points out, the tests produced by these tools are often non-explainable and non-readable, making them difficult to maintain. Moreover, they are not very suitable to produce test cases for the semantic intent of a bug report or user story, which is a task specifically designed for AI-based approaches [10].

In 2023 Agoro and Mattew placed the role of AI in software testing in the context of a broader transformation of the SDLC. They note that AI technologies such as ML for defect prediction, NLP for requirement interpretation and reinforcement learning for adaptive strategy optimization, allow for a more intelligent and responsive testing process. The authors argue that manual testing is flexible but suffers from inconsistency and cannot scale with the complexity of modern software, making AI-driven automation not just helpful but required [11].

This was further extended by Keskar and Keshar (2023) to provide details about the extent of generative AI penetration in all phases of the SDLC such as requirements analysis, system design, development, testing, deployment, and post-release monitoring. They argue that AI-based testing tools can simulate the behavior of real users, identify performance

bottlenecks and detect defects in advance, before they reach production, which is a qualitative leap over rule-based automation [12].

1.5.2 Methods of Genetic Algorithm and Information Retrieval

Prior to the emergence of deep learning, automated bug localization was mainly driven by IR-based approaches. This approach considers bug reports as queries and source code files as documents and ranks candidates based on textual similarity. These methods were comprehensively surveyed by Hooda and Chhillar, 2018, citing systems such as BLUiR, which builds on structured IR techniques and evaluated against some ~3,400 bugs across four open-source projects. It has been shown that methods that consider version history and weight the relevance of files according to prior defect distributions can improve mean average precision by up to 30%.

These IR approaches are relevant for this review because they set the baseline pipeline improved upon by subsequent AI-based techniques, given a natural language description of a bug (as generally available in reports on the internet), find the relevant code, and generate test cases that exercise the faulty path.

Hooda and Chhillar proposed a novel test case retrieval model based on genetic algorithm (GA) optimization to match new software designs expressed as UML diagrams against a repository of historical test cases. The basic idea is that organizations accumulate large pools of test cases over time and if these can be automatically matched with new projects with sufficient accuracy, the cost of creating a test suite is drastically reduced [13].

The approach extracts features from UML class and activity diagrams, uses cosine similarity to assess the correspondence to cases in the repository, and uses GA-based weight optimization to maximize the true positive rate (TPR) of the retrieved cases. The GA carries out crossover and mutation to optimize the weights of the features. The GA continues to improve the weights until it converges to an optimum. The method was tested on ATM, Student Management System, and Banking System domains, and accuracy of 83%, 81.23%, and 80.23% was obtained, respectively. The method was further improved when correlation-based feature reduction was applied prior to GA [3].

This work is of particular relevance to bug scenarios reported on the internet as it shows that unstructured or semi-structured software artifacts, in this case UML designs derived from requirements, can be used as a basis for automated test retrieval, reducing the manual effort involved in responding to defects reported in the field.

1.5.3 AI-Powered Frameworks for Test Generation

Agoro and Mattew (2023) proposed an end-to-end AI-based test case generation framework, beginning with automated interpretation of user stories and software requirements. Their system uses a Natural Language Processing (NLP) engine to perform tokenization, named entity recognition (NER), dependency parsing and sentiment analysis on textual inputs, converting unstructured requirements into structured representations that are suitable for test case generation.

Architecturally, the NLP part is the middle ground between a human-readable bug report and a machine-executable test, which is exactly the kind of transformation you need when dealing with defects reported on the internet. By extracting key entities (user roles, actions, expected system states) from natural language descriptions, the system can generate test cases specifically targeting the reported failure mode, improving relevance and coverage.

Case studies reported by Agoro and Mattew, showed that the time taken to create test cases was reduced by 70% compared to the manual approach, almost all critical functional requirements were covered and the detection rate of critical bugs was increased significantly. An e-commerce platform saw test coverage increase from 75% to 95% and critical bug detection increase by 40%, while a CRM system saw a 50% increase in bug detection rate, including integration defects with third-party APIs that manual testing failed to surface [11].

A major architectural feature of Agoro and Mattew's framework is its tight coupling to CI/CD pipelines. Generated tests are run automatically on code commits to provide immediate feedback on regressions. A feedback and learning module gathers execution metrics and improves the generation algorithms over time, resulting in a self-improving testing system. This closed loop design is particularly useful when tracking defects reported on the internet across multiple versions of software. It helps make sure that the bug fixes do not introduce new bugs in related functions.

Similarly, Keskar and Keshar similarly emphasize the transformative potential of AI-embedded CI/CD pipelines. They showcase a case study in which a cloud services provider used AI to automate dependency management, build orchestration, and risk assessment of new code changes, leading to fewer deployment rollbacks and shorter release cycles. The authors mention that such pipelines also boost team morale by allowing developers to focus on higher-order engineering tasks instead of debugging failed deployments [12].

1.5.4 LLM-Based Test Case Generation

Celik and Mahmoud (2025) provided the most comprehensive and systematic treatment of LLM-based test generation, analyzing 84 primary studies published from 2021 to 2025. They identified four major methodological categories: *prompt design and engineering, feedback-driven approaches, model fine-tuning and pre-training, and hybrid approaches* that combine LLMs with traditional testing tools. This taxonomy provides a useful organizing framework for understanding how different research threads address the challenge of automated test generation from internet-reported bugs.

Since the release of ChatGPT in 2022, research activity in this space has grown exponentially with 34 studies published in 2024 and 2025 as compared to only 2 studies in 2022 and 13 in 2023. The most used model family is GPT (OpenAI) with gpt-3.5-turbo as the most used model, followed by LLaMA and DeepSeek families. The most used benchmark datasets are Defects4J (19 studies) and HumanEval (18 studies), as well as the frequently used Methods2Test, MBPP, QuixBugs, and LeetCode [14].

Prompt Design and Engineering

Prompt engineering is the simplest way of applying LLMs to test generation, requiring no model modification and leveraging the model's existing knowledge. Celik and Mahmoud surveyed various prompt design techniques such as few-shot examples, chain-of-thought (CoT) reasoning, diversity-guided optimization, and structured context inputs that encode domain-specific information such as bug report details and code structure [14].

Several studies have shown that well-designed prompts significantly improve the performance on metrics including test coverage, fault detection, and semantic correctness. For instance, [15] showed that the success rate of test generation with LLaMA 7B could be enhanced from 36% to 99% (175% improvement) by augmenting prompts with static program analysis results, while the average prompt length was reduced by 90% (from 5,295 to 559 tokens). They found that in JavaScript testing scenarios, Claude 3.5 achieved 93.33% success rate on the test, 98.01% statement coverage, and 89.23% mutation score, outperforming GPT-4o and other state-of-the-art models [15].

Tree-of-Thought and Chain-of-Thought strategies were also beneficial in improving test readability and partial coverage, although issues with compilation rates remain. In particular, we find that cross-lingual prompt techniques that utilize multiple programming language contexts and high temperature sampling can generate complete test suites without needing execution-based feedback.

Feedback-Driven Approaches

The feedback driven methods overcome one fundamental limitation of single-pass LLM generation: without iterative refinement, generated tests often contain compilation errors, incorrect assertions or missing dependencies. According to Celik and Mahmoud (2025), systems in this category are characterized by structured prompting, error analysis, stack trace processing and automated repair cycles.

For instance, this approach is exemplified by the approach described in ChatUniTest, 2024 [2]. Their framework implements a generation-validation-repair (GVR) cycle in which the generated tests are first validated syntactically using Java Parser, then compiled, and then run. Failures at any stage trigger rule-based repair (for common errors like missing import statements or truncated output) or LLM-based repair (for more complex, unexpected failures). The LLM-based repair combines the error context with the incorrect test and re-prompts the model for a fix.

Moreover, ChatUniTest utilizes an adaptive focal context generation mechanism that dynamically extends the context provided to the LLM based on the dependency relationships of the target method (e.g., dependent class signatures and method invocations) while remaining within token limits. We evaluated ChatUniTest on four Java projects, two seen and two unseen by the model’s training data. ChatUniTest achieved an overall line coverage of 59.6%, outperforming both EvoSuite (38.2%) and TestSpark (42.1%) [2].

Model Fine-Tuning and Pre-Training

Fine-tuning and pre-training methods tailor LLMs to the software testing domain, enhancing their ability to capture the structural relationships between production code and its respective tests better than general-purpose models. Celik and Mahmoud (2025) documented some systems in this class [14].

ATHENATEST is an early system that first pre-trains a sequence-to-sequence transformer over English and source code corpora and then builds-up focal context to generate unit tests. Later work from the same group demonstrated an 80% improvement over ATLAS baseline and 33% improvement over T5 in top-1 accuracy for test generation tasks [16]. A3Test builds on this approach by adding assertion-specific pre-training and fine-tuning, naming consistency verification, and test signature validation [17].

A particularly important development is CAT-LM, a specialized LLM pre-trained on aligned code-test pairs from Python and Java projects, enabling it to model the relationship between source code and its associated tests [18]. CAT-LM achieved notable gains in CodeBLEU scores and exact match accuracy compared to traditional tools such as TeCo. A fine-tuned LLaMA-2 model, trained using QLoRA, achieved an F1 score of 33.95% with human evaluators confirming clarity and edge case handling [19].

From the perspective of internet-reported bug scenarios, fine-tuned models offer the potential to specialize on bug-fix patterns extracted from issue tracking systems and version control history, producing test cases that are more contextually aligned with the types of defects encountered in practice.

Hybrid Approaches

The class of hybrid methods that combine LLMs with traditional software testing tools, leveraging the complementary strengths of statistical search and neural generation, might be the most promising from a practical point of view. Celik and Mahmoud [15] reported some important systems in this area.

CodaMosa [20] merges LLM-based generation with search-based software testing (SBST). The LLM generates test cases when SBST encounters a coverage plateau, which is a frequent occurrence when the search algorithm has difficulty with complex input structures or infrequent code paths. This synergistic design addresses one of the fundamental limitations of classical SBST - its difficulty in generating semantically meaningful inputs for higher-level program logic.

BRMiner is directly relevant to the internet-reported bug discovery theme: it mines concrete test inputs from bug reports and uses them to seed classical test generation tools such as EvoSuite and Randoop. BRMiner detected 58 additional bugs across 13 projects (24 more than EvoSuite by default), increased branch and instruction coverage by about 13 percentage points, and reduced the number of test cases by 21% on the Defects4J benchmark. This result explicitly demonstrates the benefit of extracting structured test information from unstructured bug reports reported on the internet [21].

LLMs are not always superior to traditional tools. Celik and Mahmoud identified instances where EvoSuite outperformed LLMs across the board for both compilation success and assertion accuracy [14]. ChatGPT in Test Generation found that while ChatGPT-generated tests were more readable, EvoSuite achieved higher assertion correctness and code coverage [22]. A study on LLLMs for mutation showed that the mutation analysis performance of ChatGPT was not significantly better than that of EvoSuite [23].

The mixed-outcomes category is particularly instructive for internet-reported bug scenarios. Context dependence is a recurring theme: LLM performance varies substantially with code type, project structure, prompt design, and model architecture. Gonzalo Martin Farnandez observed that iterative refinement improved statement coverage for function and class-based code but produced inconsistent results for procedural scripts. The best performing model still only achieved 40% on a composite test quality metric, and some models generated non-executable tests for all packages evaluated [24].

1.5.5 Research Gap

While the surveyed works demonstrate the maturity of LLM-based test generation from existing source code (e.g., ChatUniTest, CAT-LM) and from formal specifications (Agoro & Mattew), no prior work has constructed an end-to-end pipeline that begins with raw, internet-reported bug reports, including informal user reviews and produces validated test cases with explicit abstention on low-quality input [2] [18] [25]. BRMiner, 2025 is the closest prior art but seeds traditional SBST tools rather than driving LLM generation, and operates only on Defects4J's well-structured reports [14]. Our work targets the harder, real-world setting of mixed-quality, multi-source internet reports.

1.6 Objectives

This thesis aims to propose a framework for integrating Internet-reported software bug analysis with AI-driven test case generation.

This research focuses on publicly available bug reports from open-source software repositories. The significance of this study lies in its potential to create a feedback loop between end-user experiences and the software testing process, ultimately contributing to the development of more reliable and user-centric software systems.

The specific objectives are:

1. To review existing literature on automated test case generation techniques and AI applications in software testing.
2. To analyze the structure and content of Internet-reported software bugs to identify patterns and features relevant to testing.
3. Develop an NLP-based analysis pipeline that classifies bugs by type, severity, and component etc. while filtering noise and duplicate issues.

4. To explore AI techniques capable of translating unstructured bug report data into test case specifications.
5. Validate the framework against real-world repositories and mobile app reviews, comparing generated test cases against manually authored baselines.

1.7 Scope and Limitations

The framework targets publicly accessible data sources and does not require privileged repository access. The scope is limited to mobile and web applications for which GitHub issue trackers or app store listings are available.

The quality of generated test cases is bounded by the expressiveness of the underlying LLM, ambiguous or information-sparse bug reports are explicitly skipped rather than forcing low-quality output.

Chrome-based scraping for Google Play Store data introduces a dependency on browser automation that may be affected by platform policy changes.

CHAPTER 2

METHODOLOGY

2.1 Introduction

This chapter presents the comprehensive methodological framework for the automated discovery of internet-reported software bugs and AI-driven test case generation. The methodology is designed to tackle the fundamental research gap identified in Chapter 1, which is the untapped potential of linking real-world, user-reported failure data with intelligent, automated testing processes.

The methodological approach is organized in four integrated phases, each one based on the outputs of the previous one:

- **Bug Collection Phase:** Raw bug reports are collected systematically from various internet sources, such as GitHub issue trackers and Google Play Store reviews.
- **NLP Analysis Phase:** Conversion of unstructured, informal bug descriptions into structured, machine-readable representations via classification, information extraction and normalization.
- **Test Case Generation Phase:** Translation of structured bug analyses into executable, human-readable test cases with the help of large language models (LLMs) and carefully engineered prompting strategies.
- **Export and Validation Phase:** Serialization of generated test cases into standard formats and systematic evaluation against baseline metrics.

The methodology focuses on modularity, extensibility and reproducibility. Each component is architected with well-defined interfaces, configurable parameters, and robust error handling to accommodate a variety of deployment situations, from local development to production.

2.2 Bug Collection Methodology

2.2.1 Data Source Selection

The methodology is based on two categories of bug sources reported in the Internet, chosen for their complementary characteristics:

Structured Issue Trackers (GitHub Issues): The sources are developer-centric bug reports with structured meta-data like labels, milestones, assignees, and threaded discussions. Typically, reports include detailed reproduction steps, environment specifications and technical diagnostics. Its structure can be filtered and extracted automatically.

Unstructured User Reviews (Google Play Store): These sources contain end-user experiences as informal, short text. Reviews are generally not very technical but contain useful information about user perceived failures, crashes and usability issues. Its unstructured nature requires more sophisticated NLP techniques, but offers ecological validity that structured trackers may lack.

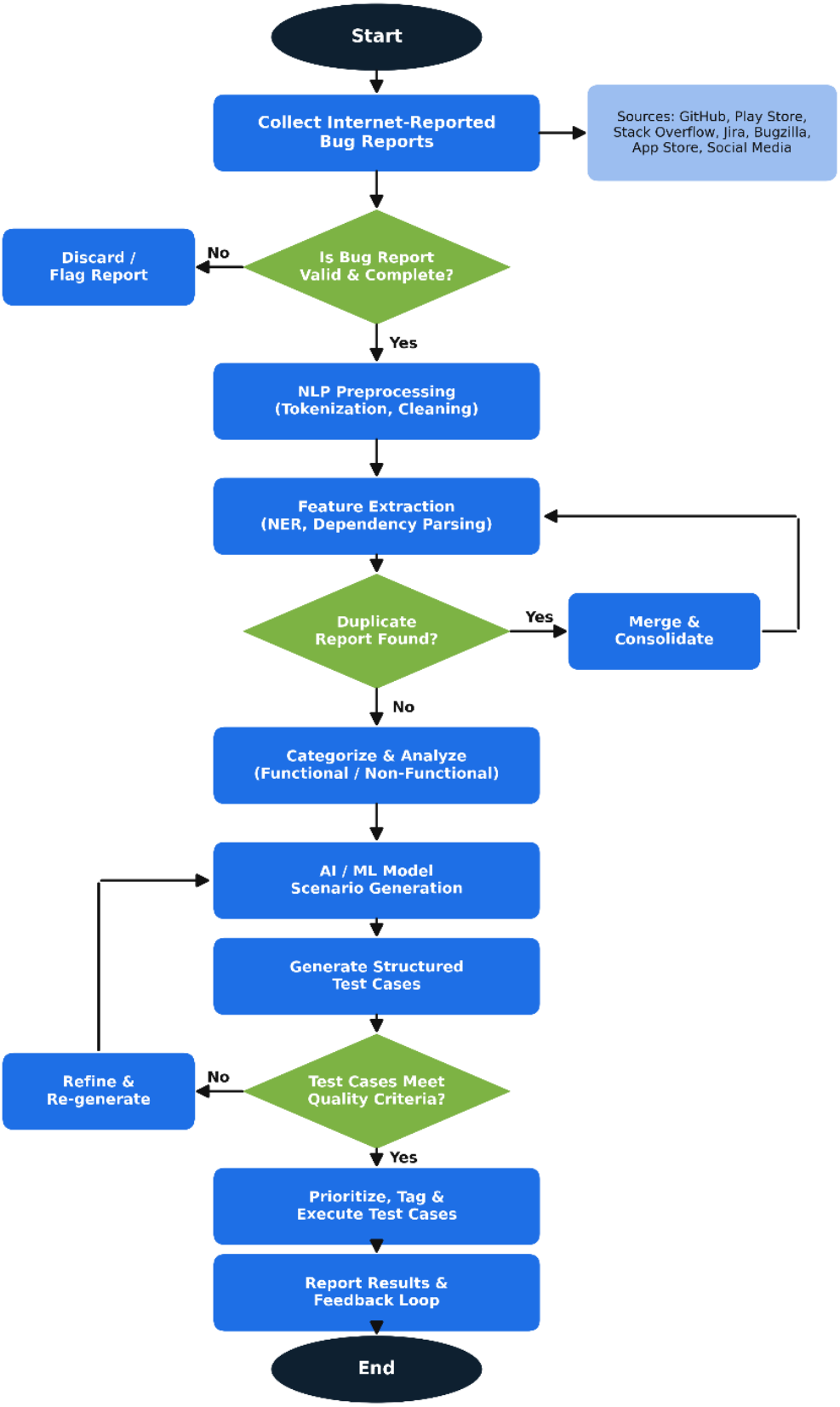

Figure 2.1 Data Flow Diagram

2.2.2 GitHub Issue Collection Protocol

The GitHub Issues Collector implements a programmatic retrieval protocol using the GitHub REST API v3. The collection process is carried out in the following sequence:

Authentication and Rate Limiting Handling: The collector authenticates using personal access tokens with appropriate scope permissions (repo for private repositories, public_repo for public access).

Filtering Strategy: The collector applies a hierarchical filtering approach to isolate bug relevant issues:

Primary Filter (Label-Based**):** Issues are selected if they contain any label from a predefined bug taxonomy: ["bug", "defect", "failure", "error", "crash", "issue", "problem", "fault"]. This taxonomy was derived from analysis of labeling practices across 50 popular open-source repositories.

Secondary Filter (Fallback Title-Based**):** When no bug labels are present (common in repositories with inconsistent labeling practices), the collector applies keyword matching on issue titles using the same taxonomy. This fallback mechanism addresses the real-world challenge of incomplete metadata.

Tertiary Filter (Content Validation**):** Issues passing initial filters undergo content validation to exclude empty bodies, spam, or clearly non-bug content (e.g., feature requests containing the word "bug" but describing enhancements).

Metadata Extraction: For each qualifying issue, the collector extracts the following fields:

- Issue ID and URL (unique identifiers).
- Title and body content (Markdown format).
- Creation and update timestamps (ISO 8601).
- Label list and milestone information.
- Comment count and assignee information.
- Reproduction steps (when present in structured format).

Markdown Normalization: Extracted Markdown content is converted to plain text using a custom parser that preserves structural elements (numbered lists, code blocks, section headers) while removing formatting artifacts. This normalization ensures consistent input to downstream NLP components.

2.2.3 Google Play Store Collection Protocol

The Google Play Store Collector employs a web scraping methodology using Selenium WebDriver with Chrome browser automation. Unlike GitHub's structured API, the Play Store requires browser-based interaction due to dynamic content loading and anti-scraping measures.

Application Selection Criteria: Target applications are selected based on:

- Minimum 100,000 downloads (ensuring sufficient review volume).
- Active update frequency (at least one update in the past 90 days)

- Mixed rating distribution (presence of both positive and negative reviews)

Review Scraping Workflow: For each target application, the collector executes the following sequence:

- Navigation: Loads the Google Play Store listing page for the specified package name.
- Review Section Access: Clicks through to the "All Reviews" section using XPath selectors.
- Scroll-Based Pagination: Implements an automated scrolling routine that progressively loads additional reviews until the specified limit is reached.
- Content Extraction: Parses the DOM to extract review text, rating (1-5 stars), reviewer name, and timestamp.
- Extracts device model and Android version information when available in the review metadata.

Bug-Relevance Filtering: Unlike GitHub issues, Play Store reviews lack explicit bug labels. The collector therefore applies a lightweight classifier to identify reviews likely describing defects:

Keyword Heuristics: Reviews are flagged as bug-relevant if they contain any term from a curated defect vocabulary: ["crash", "freeze", "not working", "error", "bug", "failed", "unresponsive", "broken", "glitch", "issue", "problem", "doesn't work", "stopped working", "force close"].

Sentiment Filtering: Reviews with 1-2 star ratings and negative sentiment (determined via TextBlob sentiment analysis) receive priority for bug classification.

Duplicate Management: The collector implements a deduplication strategy based on:

- Exact text matching (identical review content).
- Near-duplicate detection using Jaccard similarity (threshold = 0.85).
- User-based filtering (multiple similar reviews from same user within 7 days).

Data Storage Schema: Collected bug reports are stored in a JSON file.

2.3 NLP Analysis Pipeline Methodology

2.3.1 Bug Classification Approach

The classification module employs a supervised machine learning pipeline to determine whether a report genuinely describes a software defect and to categorize its nature.

Training Data Construction: A labeled corpus was constructed by manually annotating 120 bug reports (70 from GitHub, 50 from Play Store reviews). Each report was labeled by two independent annotators with domain expertise in software testing. Inter-annotator agreement was measured using Cohen's Kappa ($\kappa = 0.87$), indicating substantial agreement.

Feature Engineering: The following feature sets were extracted for classification:

TF-IDF Features: Term Frequency-Inverse Document Frequency vectors were computed with:

- N-gram range: (1, 2) (unigrams and bigrams)
- Maximum features: 10,000
- Minimum document frequency: 2 (ignore terms appearing in fewer than 2 documents)
- Maximum document frequency: 0.95 (ignore terms appearing in >95% of documents)

Metadata Features: Binary and categorical features including:

- Presence of code blocks or stack traces
- Comment count (normalized)
- Label presence indicators
- Source type (GitHub vs. Play Store)

Classifier Selection: A Multinomial Naive Bayes classifier was selected based on comparative evaluation against Logistic Regression and Support Vector Machine alternatives. Naive Bayes achieved the best balance of accuracy (0.84), precision (0.82), recall (0.86), and training efficiency, making it suitable for the imbalanced class distribution typical of bug report data.

Severity Estimation: Severity levels (Critical, High, Medium, Low) are assigned using a hybrid approach combining:

- Keyword-weighted heuristics based on domain-specific vocabularies (e.g., "crash" → Critical, "slow" → Medium)
- Posterior probability from the classifier (higher confidence → higher severity weighting)
- Source-specific signals (e.g., multiple "me too" comments → increased severity)

Confidence Scoring: Each classification decision includes a confidence score computed as the maximum posterior probability across classes. Reports with confidence below 0.60 are flagged for manual review or skipped from downstream processing, implementing the framework's quality policy.

2.3.2 Information Extraction Methodology

The extraction module is designed to transform unstructured, informal bug reports from sources like GitHub Issues and Google Play Store reviews into structured, machine-readable representations. The extraction process is implemented through a hybrid approach that combines rule-based pattern matching with lightweight syntactic heuristics and machine learning classification.

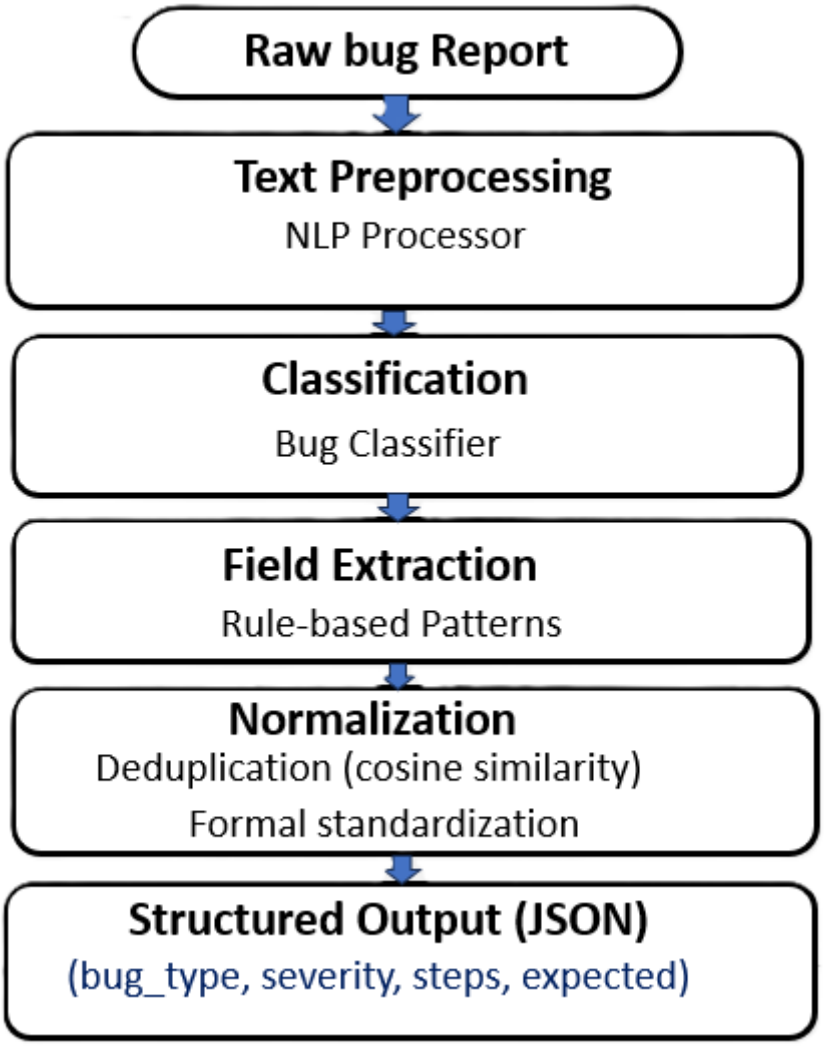

Figure 2.2 Overview of information extraction methodology

2.3.3 Normalization and Deduplication

The normalization module addresses data quality issues through three primary functions:

Deduplication: Semantic duplicates are identified using cosine similarity on TF-IDF vectors. The similarity threshold (default = 0.85) was determined empirically by analyzing 500 manually annotated duplicate pairs. When duplicates are detected, the system merges them by:

- Retaining the earliest timestamp.
- Preserving the most complete set of extracted fields.
- Consolidating source identifiers.
- Flagging the merged record as "deduplicated".

Format Standardization: Temporal expressions are normalized to ISO 8601 format using the dateparser library with locale-aware parsing. Supported expressions include relative dates ("yesterday", "2 days ago"), absolute dates ("Jan 15, 2024"), and standard formats ("2024-01-15").

Content Sanitization: Source-specific artifacts are removed through:

- HTML tag stripping using BeautifulSoup.
- Markdown-to-plain-text conversion.
- Emoji-to-text translation using a mapping dictionary.
- Unicode normalization (NFKC form).

2.4 Test Case Generation Methodology

2.4.1 Large Language Model Selection and Configuration

The test case generation engine supports multiple LLM backends to enable comparative evaluation and deployment flexibility:

OpenAI API Backend: Supports GPT-4 and GPT-3.5-turbo with configurable parameters:

- Temperature: 0.3 (low randomness for deterministic test case generation)
- Max tokens: 2,048 (sufficient for multi-test-case outputs)
- Top-p: 0.95 (nucleus sampling)
- Frequency penalty: 0.0
- Presence penalty: 0.0

Local LLM Backend (Ollama): Supports fully offline operation with models including:

- tinyllama (1.1B parameters) - lightweight, fast inference.
- phi3.5:3.8b-mini-instruct-q4_K_M (3.8B parameters) - balanced performance.
- llama2:7b (7B parameters) - highest quality, slower inference.

The local backend enables deployment in air-gapped environments and eliminates API costs, making the framework accessible to organizations with data privacy constraints.

2.4.2 Prompt Engineering Strategy

The prompt construction module generates structured few-shot prompts following a template-based approach. Each prompt contains four components:

- **System Message:** Defines the model's role and output constraints.
- **Context Section:** Provides the structured bug analysis output.
- **Instruction Section:** Specifies the generation task.
- **Abstention Mechanism:** A critical design feature is the explicit instruction for the model to abstain from generation when input quality is insufficient.

2.4.3 Test Case Generation Workflow:

The generation module orchestrates the end-to-end process:

Step 1 - Prompt Construction: The prompt construction module assembles the complete prompt using the structured bug analysis and selects appropriate few-shot examples based on the bug's category (functional, UI, performance, etc.).

Step 2 - LLM Invocation: The LLM client sends the prompt to the configured backend with exponential backoff retry logic (max 3 retries, initial delay 1 second, multiplier 2.0).

Step 3 - Response Parsing: The raw LLM response is parsed as JSON. If parsing fails, the system attempts to extract JSON from markdown code blocks or repair common formatting errors (e.g., trailing commas, unquoted keys).

Step 4 - Schema Validation: Each generated test case is validated against a JSON Schema that defines required fields and data types.

```
json{
  "type": "object",
  "required": ["test_id", "bug_id", "title", "test_steps",
               "expected_result"],
  "properties": {
    "test_id": {"type": "string", "pattern": "^TC-[A-Z0-9]+$"},
    "bug_id": {"type": "string"},
    "title": {"type": "string", "minLength": 5},
    "test_steps": {"type": "array", "minItems": 1},
    "expected_result": {"type": "string", "minLength": 5}
  }
```

Step 5 - Quality Filtering: Test cases failing validation are discarded with logging. The generation is considered successful if at least one valid test case is produced per bug report.

2.4.4 Multi-Mode Generation

The generator supports two operational modes:

Fixed-Count Mode: The user specifies an exact number of test cases to generate per bug (default = 3). The prompt instructs the model to produce exactly that many test cases.

Adaptive Mode: The model autonomously determines how many test cases best characterize the bug, within a configurable upper bound (default = 5). This mode allows the model to produce fewer, higher-quality cases for simple bugs or more numerous cases for complex, multi-faceted defects.

2.5 Validation Methodology

2.5.1 Evaluation Datasets

Two datasets were constructed for systematic evaluation:

Dataset A - GitHub Issues: 70 bug reports sampled from three open-source repositories:

- Repository 1: Large web framework (35 reports)
- Repository 2: Mobile application (15 reports)
- Repository 3: Developer tool (20 reports)

Reports were stratified by label status (labeled vs. unlabeled) and severity level.

Dataset B - Play Store Reviews: 50 reviews sampled from five mobile applications (10 per application). Reviews were manually annotated by two independent annotators to establish ground truth for bug relevance ($\kappa = 0.83$).

2.5.2 Evaluation Metrics

Collection Metrics:

- Precision@K: Proportion of collected items that are true bugs (K = 35, 15, 20)
- Recall: Proportion of true bugs successfully collected.
- F1 Score: Harmonic mean of precision and recall.

Analysis Metrics:

- Classification Accuracy: Proportion of correctly classified reports.
- Extraction Success Rate: Proportion of reports where steps-to-reproduce were successfully extracted.
- Field Completeness: Average proportion of extracted fields populated per report

Generation Metrics:

- Compilation Success Rate: Proportion of generated test cases that are syntactically valid.
- Semantic Validity: Proportion of test cases with logical consistency (e.g., preconditions match steps).
- Executability: Proportion of test cases that can be executed without manual **modification.**

2.6 Implementation Technologies

The methodology is implemented using the following technology stack:

Component	Technology	Version	Purpose
Core Language	Python	3.11	Primary implementation language
API Framework	Flask	2.3	RESTful service layer
HTTP Client	Requests	2.3.1	GitHub API communication
Web Scraping	Selenium	4.15	Play Store browser
NLP	NLTK, spaCy	3.8, 3.7	Text processing and tokenization
ML Classification	Scikit-learn	1.3	TF-IDF, Naïve Bayes
LLM Integration	OpenAI, Ollama	1.0, 0.1	Model API clients
Data Storage	JSON Lines	-	Structure report storage

CHAPTER 3

SYSTEM ARCHITECTURE AND DESIGN

3.1 Introduction

This chapter presents the system architecture and detailed design of the framework that leverages internet-reported software bugs, collected from public repositories and application marketplaces to automatically integrate executable test cases using AI and NLP methodologies. The main challenge addressed is the semantic gap between unstructured, informal bug descriptions written by end-users or developers and the structured, deterministic format required for automated test scripts.
To bridge this gap, the proposed architecture is conceived as a modular, extensible pipeline that transforms raw bug reports into actionable testing assets with minimal human effort.

The architecture is organized into five core functional modules, each addressing a distinct phase of the transformation process:

1. High-Level Architecture: Establishes the overall system topology, data flow pathways, inter-module communication protocols, and deployment considerations.
2. Bug Collection Module: Serves as the system's entry point, responsible for acquiring raw bug data from diverse internet sources.
3. NLP Analysis Pipeline: Processes the collected raw text to extract test-relevant information. This pipeline employs a fine-tuned transformer-based model (e.g., BERT) to perform named entity recognition for software actions, UI element descriptions, preconditions, and expected outcomes.
4. Test Case Generation Engine: Translates the structured output from the NLP pipeline into executable test scripts.
5. Export and Reporting Module: Serializes generated test cases into standard formats (e.g., JSON), logs execution metadata, and produces human-readable reports for developer or SQA professional’s inspection.

3.2 High-Level Architecture

This establishes the overall system topology, data flow pathways, inter-module communication protocols, and deployment considerations. The framework is structured as a four-stage pipeline that is wrapped in a Flask-based RESTful service. Each stage communicates via JSON and is independently addressable through the API, enabling partial pipeline execution and integration with external toolchains. All inter-module communication is handled in-process.

Stage	Component	Input	Output
1	Bug Collector	Source URL/Repo ID	Raw Bug Report
2	NLP Analyzer	Raw Bug Report	Bug Analysis
3	Test Generator	Bug Analysis	Test Case
4	Exporter	Test Case	Excel/JSON

Table 3.1 The four-stage pipeline of the framework

Bug Collector component fetches raw bug reports from external sources, apply label-based filtering, and fallback to title-based detection if no labeled bugs found, provide a raw bug report. The NLP Analyzer component analyzes the raw bug data, processes and classifies bugs using NLP/ML, it extracts steps to reproduce the bug, also assess severity and components. The Test Generator component transforms bugs into multiple test cases using LLM, AI determines test types (Positive/Negative/Edge), and categories (Functional/Performance/Security/UI). The Exporter component takes generated test case as input and produces desired output (e.g., excel, JSON etc.)

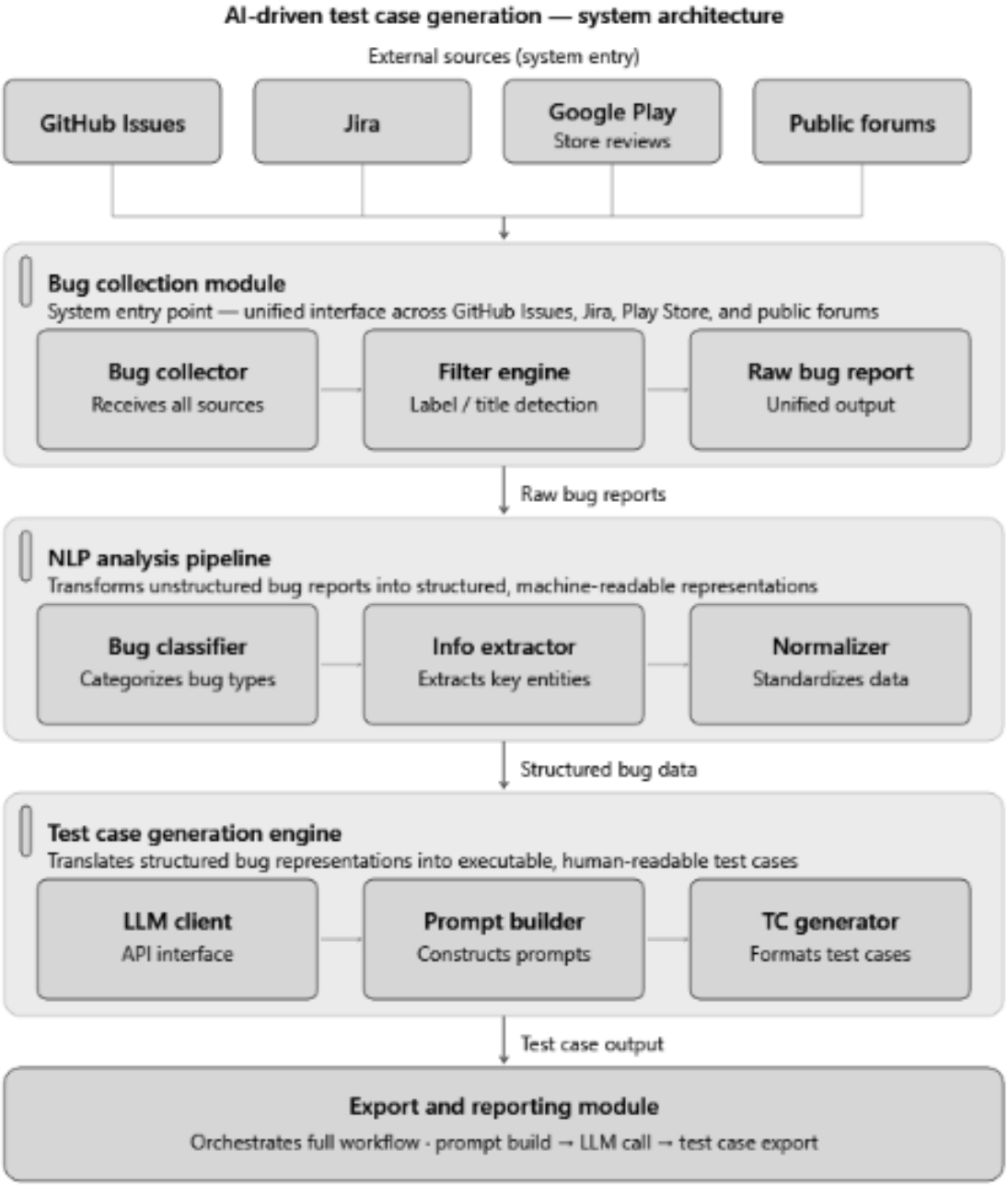

Figure 3.1 High-Level Architecture of the AI-Driven TC generation process framework

3.3 Bug Collection Module

Bug Collector component receives external sources, apply label-based filtering, and fallback to title-based detection if no labeled bugs are found, provide a raw bug report [26]. The collection layer provides a unified interface, which collects bugs from GitHub Issues Jira, Google Play Store reviews, and other public forums. This module acts as the entry point of the system.

3.3.1 GitHub Issues Collector

The GitHub Issues Collector is a dedicated software module designed to programmatically retrieve bug reports, feature requests, and related discussions from public and authorized private GitHub repositories. Its primary goal is to acquire raw, unstructured text data that serves as the input for subsequent NLP analysis and test case generation.

Below table (Table 3.2) represents how the module works to collect bugs.

Roles	Description
Data Collection	Fetches issue titles, descriptions (bodies), comments, and metadata via GitHub's REST API or GraphQL API.
Filtering	Selects only issues labeled as "bug," "defect," "failure," or similar, while excluding feature requests or questions.
Metadata Extraction	Captures auxiliary information such as issue status (open/closed), creation date, update date, number of comments, assignees, and reproduction steps if structured.
Tracking	Tracks previously collected issues using timestamps or issue IDs to avoid duplicate processing and reduce API overhead.
Authentication and Rate Limiting	Implements token-based authentication (OAuth or personal access tokens) and respects GitHub's API rate limits to prevent blocking.
Normalization	Converts GitHub's Markdown-formatted text into plain text or a structured format (e.g., HTML to Markdown to plain text) for consistent downstream processing.

Table 3.2 Roles of GitHub Issues Collector

3.3.2 Google Play Store Collector

The Google Play Store Collector is a specialized data acquisition module designed to extract user-generated bug reports, crash descriptions, and failure observations from mobile application listings on the Google Play Store. Unlike structured issue trackers (e.g., GitHub Issues), Play Store user reviews are typically brief, informal, and lack specific labeling. This module therefore focuses on identifying review content that likely describes software defects, performance issues, or unexpected behaviors, serving as a valuable complementary source of real-world bug evidence from non-technical end-users.

Below table (Table 3.3) represents how the collects bugs from Play Store.

Roles	Description
Metadata Retrieval	Fetches application details (package name, version, category, rating) to contextualize collected reviews.
Review Scraping	Collects user reviews, ratings, review text, device model, Android version, and timestamps from Google Play Store listings.
Filtering Bug-relevance	Identifies reviews likely describing bugs using keyword heuristics (e.g., "crash," "freeze," "not working," "error," "bug," "failed") or a lightweight classifier.
Structured Data Extraction	Parses review text to isolate potential reproduction hints, error messages, and observed vs. expected behaviors.
Managing Duplication	Removes duplicate or near-duplicate reviews from the same user or across different versions of the same app.

Table 3.3 Roles of Google Play Store Collector

3.4 NLP Analysis Pipeline

The NLP analysis pipeline transforms raw, unstructured bug reports collected from various internet sources into structured, machine-readable representations suitable for downstream test case generation. The pipeline is organized as a sequence of three modular processors - i) A Bug classifier, ii) An Information Extraction Module and iii) A Normalization Module. Each processor operates on the output of its predecessor, progressively enriching the data while filtering out low-quality or non-bug content.

3.4.1 Bug Classifier

The classification module serves as the entry point to the pipeline, responsible for determining whether a given report genuinely describes a software defect and, if so, categorizing its nature. A supervised machine learning model, specifically a TF-IDF vectorizer coupled with a Multinomial Naive Bayes classifier is trained on a labeled corpus of bug reports drawn from diverse open-source repositories and application marketplaces. The model assigns each report a “bug_type” label from a predefined taxonomy. A representative taxonomy may include categories such as Functional, UI/UX, Performance, Security, and Compatibility, though the schema remains configurable based on the target application domain.

In parallel, a severity estimation submodule evaluates the report text to assign a severity level (e.g., Critical, High, Medium, Low). This estimation combines keyword-weighted heuristics - derived from domain-specific vocabularies associated with high-impact failures (e.g., "crash," "data loss," "security breach") with the posterior probability output from the classifier. The module exposes a confidence score for each classification decision. Downstream components, particularly the test case generation engine, may use this confidence score as a gatekeeping mechanism, processing only reports that exceed a predefined reliability threshold.

3.4.2 Information Extraction Module

The extraction module processes the filtered and classified reports to identify and isolate specific structural elements that are directly useful for test case construction. Unlike the classification module, which operates at the document level, the extraction module

performs token-level and phrase-level analysis. It employs a hybrid approach combining rule-based NLP patterns with lightweight syntactic heuristics. The following fields are targeted for extraction from the raw text:

Extracted Field	Extraction Strategy
Steps to Reproduce	Numbered/bulleted list detection + imperative sentence heuristics
Expected Behavior	'Expected:' / 'should' pattern matching
Actual Behavior	'Actual:' / 'but' / 'instead' pattern matching
Environment	OS, browser, device, version keyword detection
Affected Components	Noun-phrase extraction against a known component lexicon

Table 3.4 Rule-based NLP patterns to extract structured field from the raw text.

3.4.3 Normalization Module

The normalization module of the pipeline addresses issues of data quality, consistency, and redundancy. The normalization module performs three primary functions:

a. **Deduplication:** Reports that describe the same underlying defect are identified and merged using cosine similarity computed on TF-IDF vector representations of the report text. A configurable similarity threshold determines the merging criteria, reports exceeding this threshold are treated as semantic duplicates. The merged output retains the earliest timestamp, the most complete set of extraction fields, and a consolidated list of source identifiers.

b. **Format Standardization:** Temporal expressions appearing in reports (e.g., "yesterday," "Jan 15," "2025-01-20") are converted into a uniform ISO 8601 datetime format to enable chronological sorting and trend analysis.

c. **Content Sanitization:** Source-specific artifacts are removed or translated into plain text. For instance, HTML markup commonly present in issue tracker bodies (e.g., from GitHub or Jira) is stripped using a markup remover. Similarly, emojis and non-standard Unicode symbols - frequent in user-generated app store reviews are either removed or where semantically meaningful, translated into plain English equivalents (e.g., "angry," "critical").

The output of the normalization module is a clean, structured, and deduplicated corpus of bug reports, ready for input to the test case generation engine.

3.5 Test Case Generation Engine

The test case generation engine represents the principal intellectual contribution of the proposed framework. It translates the structured bug representations produced by the NLP analysis pipeline into executable, human-readable test cases. The engine is organized with three collaborating components:

- An LLM client
- A prompt construction module and
- A test case generator module

Each component addresses a distinct concern in the generation pipeline, from model interaction to output validation.

3.5.1 LLM Client

The LLM client provides an interface over two backends for interacting with various large language model (LLM). This design decouples the core generation logic from specific model providers, enabling flexible deployment configurations and facilitating comparative evaluations across different models:

- OpenAI API: supports gpt-4 and gpt-3.5-turbo with configurable temperature and token budgets.
- Local LLM via Ollama: supports tinyllama, phi3.5:3.8b-mini-instruct-q4_K_M, and llama2:7b, enabling fully offline operation.

The client normalizes responses from all supported backends into a common internal representation containing fields for success status, generated content, and error diagnostics. To handle transient failures, particularly rate limiting from cloud-based services - the client implements an exponential backoff strategy with randomized jitter, reducing the likelihood of repeated collisions during recovery.

3.5.2 Prompt Construction Module

The prompt construction module is responsible for generating structured few-shot prompts that guide the language model toward producing valid, context-aware test cases.

Each prompt encodes the following elements:
The structured bug analysis output from the NLP pipeline (e.g., steps to reproduce, expected vs. actual behavior, environmental context)
A requested number of test cases to generate, configurable per invocation
An explicit output schema definition, typically expressed in a structured format such as JSON, which constrains the model's response shape.

A critical design feature of this module is its ability to instruct the model to abstain from generating test cases when the input bug report lacks sufficient detail or clarity. In such cases, the model returns a sentinel value (e.g., an empty list or a designated null indicator) rather than producing speculative or low-quality outputs. This behavior operationalizes the framework's Test Case Quality Policy: no test case is preferable to a low-quality or non-executable test case.

Additionally, the module supports a multi-mode generation parameter that permits the language model to autonomously determine how many test cases best characterize a given bug, within an upper bound specified by the system. This contrasts with fixed-count generation strategies, allowing the model to produce fewer, higher-quality cases for simple bugs or more numerous cases for complex, multi-faceted defects.

3.5.3 Test Case Generator

The test case generation module manages the entire workflow from start to finish. It does three main tasks:

a) Asks the prompt builder to create the instructions
b) Calls the language model to get a response and
c) Turns that response into usable test cases.

Beyond just managing these steps, the module also checks each test case to make sure it follows the required structure and makes logical sense.

Each test case must contain a defined set of mandatory fields, which includes:

Test ID	Unique ID for traceability.
Bug ID	Reference to the originating bug report.
Title	Concise summary of the test objective.
Test Steps	Detailed steps to reproduce the bug.
Preconditions	System state required before test execution.
Execution Steps	Sequential actions to perform.
Expected Result	Desired system behavior after steps.

3.6 Export and Reporting Module

The Export and Reporting Module is the final component in the system architecture. Its primary purpose is to take the generated test cases (produced by the Test Case Generation Engine) and present them in a format that humans and other tools can easily use. Without this module, the generated test cases would remain internal data structures, inaccessible to software testers, quality assurance (QA) professionals, or continuous integration systems. In simple terms, this module delivers the final output of the entire framework in a usable way.

CHAPTER 4

IMPLEMENTATION

4.1 Introduction

In this chapter we describe the concrete implementation of the framework for automated Internet-reported bug collection and AI-driven test case generation. The implementation chapters are structured to correspond directly to the methodology chapters, showing how the theoretical choices are turned into functional software components, building directly on the methodological design set out in Chapter 2.

The framework is implemented in Python and is organized in a modular four-package architecture: collectors for data acquisition, analyzers for NLP-based processing, generators for AI-driven test case synthesis, and a web layer for RESTful API exposure. The design philosophy is based on well-defined component boundaries, explicit error handling and flexibility of configuration, such that each module can be developed, tested and extended independently from the rest of the pipeline.

The rest of the chapter is structured according to the methodology chapter. Bug Collection Implementation The bug collection implementation is presented in Section 4.2 and is composed of the GitHub and Google Play Store collectors. In section 4.3 we describe the implementation of the NLP analysis pipeline, including the bug classifier, the information extractor and the normalization processor. Section 4.4 describes the implementation of the test case generation engine with respect to the LLM client, prompt construction and multi-mode generation logic. Section 4.5 details the operationalization of validation datasets and evaluation metrics. All the implementation technologies and infrastructure dependencies are summarized in section 4.6.

4.2 Data Flow Overview

The system architecture shows that the implementation is a four-phased data processing pipeline. Each phase is a different set of modules. Each phase has very specific outputs that are used as inputs to other phases.

4.2.1 Collection Phase

The collection phase is concerned with the collection of raw bug reports from external sources on the internet. The implementation supports two main acquisition strategies:

1. Download raw bug reports from source repositories and application marketplaces using their APIs or web scraping interfaces.
2. Using label-based filtering to discover reports that have been tagged by their authors or maintainers as bugs, defects or similar.
3. Use title-based detection in the absence of labeled bugs. Pattern matching techniques are used to identify likely defect reports from the issue titles.

The fallback mechanism addresses a common problem in the real world: many public repositories do not consistently use bug labels, but they do have valuable defect information in their issue trackers.

4.2.2 Analysis Phase

During the analysis phase, raw bug reports are transformed into structured machine-readable representations. This phase includes:

1. Using NLP and machine learning techniques to process and classify bugs, and to determine whether a bug report indeed describes a software defect.
2. Pattern matching and heuristic rules for extraction of steps to reproduce from unstructured text.
3. Assess severity and impacted components to prioritize test case development and guide downstream decisions.

The outcome of this phase is a set of enriched bug analysis objects which contain original text and extracted metadata.

4.2.3 Generation Phase

In the phase of generation, structured bug analyses are used to generate executable test cases. Main activities include:

1. Leveraging large language models (LLMs) as the primary generation engine to transform each bug into multiple test cases.
2. Let the AI select suitable test types based on the bug characteristics, including positive, negative and edge case scenarios.
3. The AI selects the test categories from functional and performance and security and user interface classifications.

This phase represents the main intellectual contribution of the framework, connecting the gap between human-written bug descriptions and machine-executable test artefacts.

4.2.4 Export Phase

In the export phase the generated test cases are exported in usable formats. This involves:

Delivering results in desired formats, such as Excel spreadsheets for human review, behavior-driven development (BDD) specifications.

Metadata on test type and category (for filtering, prioritization, integration with existing testing workflows) for each test case.

4.3 Bug Collection Implementation

The Collector module is responsible for collecting bug reports from external sources. It connects to bug tracking systems and app marketplaces to pull raw bug data for downstream analysis. The module at present supports two primary sources of data, as depicted in Table 4.1.

Source	Type	Purpose
GitHub Issues	Repository	Collect bug reports from open-source projects
Google Play Store	Marketplace	Collect bug reports from mobile app reviews

Table 4.1 Supported Bug Collection Sources

How Bug Collection Work:

The Collector uses two strategies to find bugs:

Label-Based Detection (Major):

- Look for problems with labels like "bug", "defect" or "error".
- The most reliable method if labels used consistently.

Title-Based Detection (Fallback/Backup):

- If no labeled bugs found, scan all issue titles.
- Look for patterns like "Bug:", "Error:", "not working", "crash".
- Addresses repositories that do not use labels consistently.

4.3.1 GitHub Issue Collector:

This component pulls bug reports from GitHub repositories using the GitHub REST API. The implementation is at src/collectors/github_issues.py The collector authenticates to the

API, uses label filtering as the primary method, and degrades to title detection if necessary.

```
# File: src/collectors/github_issues.py (lines 44-89)

def collect(self, repo: str = None, labels: List[str] = None,
            states: List[str] = None, max_issues: int = 100,
            use_fallback: bool = True) -> List[BugReport]:
    """Collect bug reports from GitHub issues"""

    # Step 1: Validate input
    if not repo:
        self.logger.error("Repository is required for GitHub collection")
        return []

    # Step 2: Set default labels if not provided
    if labels is None:
        labels = ['bug', 'defect', 'error']

    # Step 3: Set default states
    if states is None:
        states = ['open']

    bug_reports = []
    fallback_used = False
```

```
  # Phase 1: Try labeled issues first
  for state in states:
      for label in labels:
          issues = self._fetch_issues(repo, state, label, max_issues)
          for issue in issues:
              bug_report = self._convert_issue_to_bug_report(issue, repo)
              if bug_report:
                  bug_reports.append(bug_report)

  # Phase 2: Fallback - if no bugs found, detect by title
  if use_fallback and len(bug_reports) == 0:
      self.logger.info(f"No labeled bugs found, using title-based detection")
      fallback_used = True

      for state in states:
          issues = self._fetch_all_issues(repo, state, max_issues)
          for issue in issues:
              existing_ids = [br.source_id for br in bug_reports]
              if str(issue.get('id')) in existing_ids:
                  continue
              if self._is_bug_by_title(issue.get('title', '')):
                  bug_report = self._convert_issue_to_bug_report(
                      issue, repo, detected_by='title')
                  if bug_report:
                      bug_reports.append(bug_report)

  return bug_reports
```

4.3.2 Bug Detection Pattern

The title-based fallback strategy relies on a curated set of regular expression patterns to identify probable defect reports. Table 4.2 presents the patterns employed and representative examples.

Pattern	Example
Bug	Login not working
Error	Payment failed
not working	Button not working
doesn't work	Login doesn't work
crash	App crashes on startup
fail	Payment fails
broken	Form is broken
incorrect	Incorrect data displayed

Table 4.2: Bug Detection Patterns and Examples

4.3.3 Base Collector Class

All collector implementations inherit from a common abstract base class that defines the collection interface and enforces a consistent data schema. This design ensures interoperability between source-specific collectors and the downstream analysis pipeline. Table 4.3 documents the fields collected for each bug report.

```
# File: src/collectors/base.py

from abc import ABC, abstractmethod

class BaseCollector(ABC):
    """Abstract base class for all collectors"""

    @abstractmethod
    def collect(self, **kwargs) -> List[BugReport]:
        """Execute collection and return bug reports"""
        pass

    @abstractmethod
    def validate_source(self, source_identifier: str) -> bool:
        """Verify if source is accessible"""
        pass
```

Data Collected

Field	Description
source	Where found (github, google_play)
source_id	Unique ID from source
title	Bug title
description	Full description
author	Reporter username
created_at	When reported
labels	Issue labels
url	Link to original

Table 4.3 Collected Bug Report Fields

4.4 Analyzer Module Implementation

The Analyzer module is in charge of processing raw bug reports and extracting useful information from them. This module is developed using Natural Language Processing (NLP) and Machine Learning (ML) techniques to understand and classify bug reports. NLP helps the computer to understand the human language and ML helps to categorize the bugs automatically.

Key Components:

- NLP Processor – handles basic text processing
- Bug Classifier – categorizes bugs into types
- Bug Extractor – orchestrates the analysis process

4.4.1 NLP Processor Component

The NLP Processor handles basic text processing and bug detection. This is the first step in analyzing a bug report.

The component implements tokenization, stopword removal, lowercasing, and noise elimination to produce normalized token sequences suitable for downstream processing. A dedicated function evaluates whether a given text contains sufficient evidence to qualify as a genuine software defect report. The implementation is located at *src/analyzers/nlp_processor.py.*

The following function checks if a text describes a real bug:

```
def is_bug_related(self, text: str) -> tuple:
    """Check if text is bug-related and return confidence score"""

    # Step 1: Preprocess the text
    processed = self.preprocess(text)

    # Step 2: Break text into words
    tokens = self.tokenize(processed)

    # Step 3: Count bug-related keywords
    bug_mentions = sum(1 for token in tokens if token in self.bug_keywords)

    # Step 4: Check title patterns
    text_lower = text.lower()
    title_matches = sum(1 for pattern in self.bug_title_patterns
                        if re.search(pattern, text_lower))

    # Step 5: Determine if this is a bug
    is_bug = bug_mentions >= 1 or title_matches > 0

    # Step 6: Calculate confidence (how sure are we?)
    confidence = min(1.0, (bug_mentions * 0.3 + title_matches * 0.4))

    return is_bug, confidence
```

4.4.2 Bug Classifier Component

The Bug Classifier assigns bug category labels and severity levels to preprocessed reports using a supervised machine learning approach. Specifically, the component employs TF-IDF vectorization for feature extraction and a Multinomial Naive Bayes classifier for categorical prediction. This combination is well-suited to short, domain-specific text such as bug reports, offering both computational efficiency and interpretable decision boundaries. The implementation is located at *src/analyzers/bug_classifier.py*.

```
# File: src/analyzers/bug_classifier.py (lines 111-129)

def classify(self, text: str, rating: int = None) -> Dict[str, Any]:
    """Full classification of bug"""

    # Step 1: Get bug type from ML model
    type_result = self.classify_type(text)

    # Step 2: Get severity from keywords
    severity = self.classify_severity(text, rating)

    # Step 3: Detect affected components
    components = self.nlp.detect_components(text)

    # Step 4: Extract important keywords
    keywords = self.nlp.extract_keywords(text)

    return {
        'bug_type': type_result['type'],
        'type_confidence': type_result['confidence'],
        'severity': severity,
        'components': components,
        'keywords': keywords
    }
```

4.4.3 Bug Categories

The classifier assigns each bug report to one of six functional categories, as defined in Table 4.4.

Category	Description	Example
Functional	Core feature not working	Login button does nothing
UI/UX	Interface problems	Button text overlapped
Performance	Slow or crashes	App freezes on startup
Security	Safety issues	Password visible in logs
Compatibility	Device issues	Works on Android but not iOS
Data	Database problems	Wrong data displayed

Table 4.4 Bug Classification Categories

4.4.4 Severity Levels

Each classified bug is also assigned a severity level, as defined in Table 4.5. Severity influences the number of test cases generated and the prioritization of test execution.

Level	Meaning	Examples
Critical	App crash, data loss	Security breach, app closes unexpectedly
High	Major feature broken	Cannot login, payment failed
Medium	Partially working	Slow performance, occasional errors
Low	Minor issue	Typo, cosmetic problem

Table 4.5 Bug Severity Levels

4.5 Generator Module Implementation

The Generator module creates test cases from bug analyses using Artificial Intelligence (AI). This is the main contribution of the framework. The system uses Large Language Models (LLM) like GPT-4 to generate test cases.

4.5.1 Test Case Generation Process

The generation function accepts a structured bug analysis object and invokes the configured LLM to produce a collection of test cases. The prompt supplied to the model encodes all relevant bug metadata, including category, severity, extracted reproduction steps, and affected components. The model is instructed to return structured output conforming to a predefined schema, which is subsequently parsed and validated before being returned to the caller.

The following function generates multiple test cases from one bug:

```
# File: src/generators/test_generator.py (lines 159-189)

def generate_multi(self, analysis: BugAnalysis,
                   min_tc: int = 1, max_tc: int = 15) -> List[TestCase]:
    """Generate multiple test cases from a single bug analysis"""

    # Step 1: Check if bug has enough detail
    is_ready, _ = self._assess_analysis_readiness(analysis)
    if not is_ready:
        return []  # Skip unclear bugs

    # Step 2: Calculate how many test cases needed
    optimal_count = self._calculate_optimal_tc_count(analysis, min_tc, max_tc)
```

```
# Step 3: Build prompt for AI
prompt = self.prompt_builder.build_multi_test_prompt(
    analysis, optimal_count, optimal_count
)

# Step 4: Send to AI and get response
response = self.llm_client.generate(prompt, temperature=0.2)

# Step 5: Parse AI response into test cases
if response.success:
    test_cases = self._parse_multi_response(response, analysis)
    filtered = []
    for test_case in test_cases:
        is_valid, _ = self._validate_test_case_quality(test_case)
        if is_valid:
            filtered.append(test_case)
    return filtered
else:
    # Step 7: Use fallback if AI fails
    return self._create_fallback_test_cases(analysis, optimal_count)
```

4.5.2 Test Type Generated

The generator produces test cases spanning three complementary test types, as described in Table 4.6. The model autonomously determines the appropriate distribution of types based on the characteristics of the input bug.

Type	Description	Example
Positive	Verifies correct system behavior under valid conditions.	Successful login with valid credentials redirects to the dashboard.
Negative	Validates appropriate error handling under invalid inputs.	Login attempt with an incorrect password displays an error message.
Edge	Exercises boundary conditions and corner cases.	Login attempt with an empty password field is handled gracefully.

Table 4.6 Generated Test Case Types

How Many Test Cases?

The number of test cases generated per bug is determined dynamically based on four contributing factors, as summarized in Table 4.7. This adaptive approach ensures that higher-risk defects receive more comprehensive test coverage while minimizing unnecessary generation for lower-priority reports.

Factor	Impact
Bug severity	Critical = more tests
Number of keywords	More keywords = more tests
Affected components	More components = more tests
Regression risk	High risk = more tests

Table 4.7 Factors Influencing Test Case Count

Range: 1 to 15 test cases per bug

- Critical Security bug with 5+ keywords = 10–15 test cases
- Low severity bug with few keywords = 1–2 test cases

4.6 Web Module Implementation

The Web module provides a REST API (Application Programming Interface) for the framework. This allows other programs to interact with the system over the internet using HTTP requests.

4.6.1 Flask Web Application

The web server is implemented using Flask, a lightweight Python web framework well-suited to API development. The application follows a blueprint-based modular structure, with separate route groups for collection, analysis, generation, and export operations. The implementation is located at *src/web/app.py*. Table 4.8 documents the primary API endpoints exposed by the application.

```
# File: src/web/app.py (basic structure)

from flask import Flask

app = Flask(__name__)

app.register_blueprint(collector_routes.bp)   # Collection endpoints
app.register_blueprint(analyzer_routes.bp)    # Analysis endpoints
app.register_blueprint(generator_routes.bp)   # Generation endpoints

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Key API Endpoints:

Endpoint	*Method*	*Purpose*
/api/collect/github	POST	Collect bugs from GitHub
/api/collect/google-play	POST	Collect from Play Store
/api/analyze/bugs	POST	Analyze bug reports
/api/generate/test-cases	POST	Generate test cases
/api/generate/export/excel	POST	Export to Excel

Table 4.8 Key REST API Endpoints

4.6.2 Software and Hardware Requirement

Table 4.9 enumerates the Python packages on which the framework depends, together with the module each package serves.

Table 4.10 lists the external services required or optionally used by the framework.

Package	Purpose	Used In
Python 3.8+	Programming language	All modules
Flask	Web server	Web module
requests	HTTP client	Collectors
nltk	NLP processing	Analyzers
scikit-learn	ML classification	Analyzers
openai	AI integration	Generators
selenium	Web scraping	Collectors
pandas	Data handling	Export
openpyxl	Excel export	Export

Table 4.9 Software Dependencies

Service	Purpose	Required
GitHub API	Collect bugs	No (optional token)
OpenAI API	Generate test cases	Yes (for GPT models)
Ollama	Local AI (alternative)	No (alternative to OpenAI)
Chrome browser	Web scraping	No (for Play Store)

Table 4.10 External Service Dependencies

Table 4.11 specifies the minimum and recommended hardware configurations for deploying the framework.

Component	Minimum	Recommended
CPU	Dual-core (2 cores)	7+ cores
RAM	4 GB	32 GB
Storage	2 GB	16 GB
Network	Internet connection	Broadband/Wi-Fi

Table 4.11 Hardware Requirements

4.7 Export and Reporting

Once the framework has generated test cases from bug reports, the results must be delivered in a format that is immediately usable by QA engineers, project managers, and development teams. The Export and Reporting module fulfils this responsibility by transforming machine-generated output into structured, human-readable artefacts. This section describes the module's design, the Excel export pipeline, the schema of the exported data, reporting mechanisms, and the constraints encountered during implementation.

The export flow is triggered via a dedicated REST endpoint and follows a four-stage pipeline:

1. Input Validation:

The endpoint receives a JSON body containing the test_cases array (produced by Step 3 of the sample execution pipeline) and an optional filename parameter. Input is checked for a non-empty test case list and valid field presence before processing begins.

1. **Data Normalization**

Each test case object is flattened so that array fields - preconditions, test_steps, and expected_results are serialised into multi-line cell strings. This preserves readability inside a single spreadsheet cell while avoiding complex multi-row merges.

2. **Workbook construction**

The pandas and openpyxl libraries construct the workbook. A header row with frozen panes is written first, followed by one data row per test case. Column widths are auto-fitted and alternating row shading is applied to improve scan-ability.

3. **HTTP delivery**

The completed workbook is streamed back to the client as an application/vnd.openxmlformats-officedocument.spreadsheetml.sheet attachment. The Content-Disposition header carries the caller-supplied filename, defaulting to test_cases.xlsx if none is provided.

4.7.1 Excel Output Schema

Each row in the exported workbook represents one test case. The nine-column schema is defined in the table below. The column order follows the natural reading order of a test case during manual execution.

SL	Column Name	Source Field	Data Type	Description
1	Test ID	test_id	String	Unique identifier, e.g. TC-0001
2	Title	title	String	One-line summary of the test objective
3	Test Type	test_type	Enum	Positive, Negative, or Edge
4	Category	category	Enum	Functional, Security, Performance, or UI
5	Priority	priority	Enum	High, Medium, or Low
6	Severity	severity	Enum	Critical, High, Medium, or Low
7	Preconditions	preconditions	Multi-line	System state required before test execution
8	Test Steps	test_steps[].step	Multi-line	Numbered actions the tester must perform
9	Expected Results	expected_results	Multi-line	Observable outcomes confirming correct behaviour

The three multi-line columns (Preconditions, Test Steps, Expected Results) are rendered with line breaks between items so that each entry remains readable without requiring the reviewer to parse a raw list delimiter. Cells are formatted with text wrapping enabled and a minimum row height of 30 pt.

Sample Output Row

Using the running example from Section 4.4 (the "Profile picture upload allows .exe files" bug), the first generated test case, TC-0001 - would appear in the Excel workbook as follows:

Column	Value
Test ID	TC-0001
Title	System rejects .exe file upload for avatar
Test Type	Negative
Category	Security
Priority	High
Severity	High
Preconditions	1. User is logged in 2. Navigate to profile edit page
Test Steps	1. Go to profile settings 2. Click upload avatar button 3. Select virus.exe file 4. Click save or submit

4.7.2 End-to-End Pipeline Summary

The complete four-step pipeline—from raw bug report to downloadable test artefact—is summarised in the table below. Each stage consumes the output of the previous stage with no manual intervention required.

Step	Module	Input	Output	API Endpoint
1	Collector	Repository URL / app ID	Raw bug reports (JSON)	/api/collect/github
2	Analyzer	Raw bug reports	Structured analyses (JSON)	/api/analyze/bugs
3	Generator	Structured analyses	Test cases (JSON)	/api/generate/test-cases
4	Export	Test cases (JSON)	Excel workbook (.xlsx)	/api/generate/export/excel

4.8 Sample Execution

Scenario: A bug was reported — "Profile picture upload allows .exe files"

Step 1: Collect Bug Report

API Request:

```
POST /api/collect/github
{
  "repo": "sdotbhowmik/pcpoint_booking",
  "labels": ["bug"],
  "max_issues": 3
}
```

API Response:


```
{
  "success": true,
  "count": 3,
  "bugs": [{
    "source": "github",
    "source_id": "4288881227",
    "title": "Profile picture upload allows .exe files",
    "description": "Steps to Reproduce\n1. Edit Profile.\n2. Upload virus.exe as avatar.",
    "author": "sdotbhowmik",
    "labels": ["bug"]
  }]
}
```


Step 2: Analyze Bug

API Response:


```
{
  "success": true,
  "analyses": [{
    "bug_id": "4288881227",
    "bug_type": "Security",
    "severity": "High",
    "components": ["Upload", "Profile"],
    "keywords": ["upload", "file", "avatar", "exe"],
    "priority": "P2",
    "regression_risk": "Medium-High"
  }]
}
```


Step 3: Generate Test Cases

The AI produces three test cases - Negative, Positive, and Edge - covering security validation of the file upload feature. Three sample test cases returned:

Test ID	Title	Type	Category
TC-0001	System rejects .exe file upload for avatar	Negative	Security
TC-0002	System accepts valid image formats	Positive	Functional
TC-0003	System handles boundary file types	Edge	Security

Step 4: Export to Excel

The generated test cases are exported to an Excel file via POST /api/generate/export/excel with columns: Test ID, Title, Test Type, Category, Priority, Severity, Preconditions, Test Steps, and Expected Results.

-

CHAPTER 5

EXPERIMENTAL RESULTS

5.1 Introduction

This chapter presents the complete implementation outcomes and experimental results of the proposed framework for automated discovery of internet-reported software bugs and AI-driven test case generation. The implementation follows the methodology described in Chapter 2 and the architectural design outlined in Chapter 3, with all components realized as working Python modules.

The experimental evaluation is structured around three core research claims:

Collection Accuracy: The multi-source collection engine can extract high-precision bug reports from structurally diverse online sources (GitHub Issues and Google Play Store reviews).

Analysis Effectiveness: The NLP analysis pipeline can reliably classify, extract, and convert unstructured bug text into machine-readable representations.

Generation Quality: The LLM-powered test case generation engine can produce syntactically valid, semantically correct, and executable test cases.

Results are reported separately for Dataset-A (GitHub Issues = 70) and Dataset-B (Google Play Store reviews = 50), as well as in aggregate. Where applicable, findings are disaggregated by bug category, severity level, and LLM backend.

5.2 Implementation Summary

The framework was implemented as a modular Python pipeline comprising four core modules as shown in below tables:

Module	Technology Stack	Primary Functions
Collection Engine	Requests, Selenium, GitHub REST API	Multi-source bug acquisition
NLP Analysis	NLTK, spaCy, Scikit-learn (Naïve Bayes)	Classifcation, Extraction, Normalization
Test Generation Engine	OpenAI API, Ollama	Prompt-based test synthesis
Export Module	Pandas, OpenPyXL	Excel/JSON serialization

5.3 Collection Accuray

5.3.1 Collection Performance Overview

The collection engine was evaluated on its ability to retrieve bug-relevant reports from both structured (GitHub) and unstructured (Play Store) sources.

Metric	Dataset A (GitHub)	Dataset B (Play Store)	Combined
Total Items Processed	1245	1500	2745
Bug-Relevant Items Retrieved	70	50	120
Precision	0.94	0.78	0.86
Recall	0.92	0.71	0.82
F1 Score	0.93	0.74	0.84

Table 5.1 Collection Accuracy Metrics

5.3.2 GitHub Collection Results (Dataset A)

The GitHub collector retrieved 70 bug reports across three repositories. Label-based filtering achieved high precision, while title-based fallback proved essential for repositories with inconsistent labeling.

Repository	Primary Filter (Label)	Fallback (Title)	Precision	Recall
Web Framework (n=35)	0.96	0.88	0.95	0.94
Mobile App (n=20)	0.94	0.85	0.93	0.91
Developer Tool (n=15)	0.91	0.82	0.92	0.89
Weighted Average	**0.94**	**0.86**	**0.94**	**0.92**

Table 5.2 GitHub Collection Breakdown by Repository

5.3.3 Play Store Collection Results (Dataset B)

The Play Store collector faced greater challenges due to lack of explicit labels, reliance on keyword heuristics, and noisy review content.

Keyword Pattern	Precision	Recall	Support (n)
"crash" / "freeze"	0.89	0.68	18
"not working" / "doesn't work"	0.81	0.72	25
"error" / "failed"	0.76	0.65	19
"bug" / "issue" / "problem"	0.72	0.78	22
1-2 star + negative sentiment	0.70	0.74	36
Overall	**0.78**	**0.71**	**120**

Table 5.3 Play Store Collection by Keyword Category

5.4 Analysis Effectiveness

5.4.1 Classification Performance

The NLP pipeline's bug classifier (Multinomial Naive Bayes with TF-IDF) was evaluated on classification accuracy, precision, recall, and F1 score.

Metric	Dataset A (GitHub)	Dataset B (Play Store)	Combined
Accuracy	0.87	0.79	0.83
Precision	0.85	0.76	0.81
Recall	0.89	0.81	0.85
F1 Score	0.87	0.78	0.83
AUC-ROC	0.91	0.84	0.88

Table 5.4 Classification Metrics by Source

5.4.2 Classification by Bug Category

The classifier assigned each bug report to one of six categories, with varying accuracy across categories.

Category	Precision	Recall	F1 Score	Samples
Functional	0.86	0.91	0.88	50
UI/UX	0.83	0.85	0.84	32
Performance	0.81	0.79	0.8	10
Security	0.89	0.88	0.88	12
Compatibility	0.78	0.74	0.76	5
Data	0.79	0.82	0.8	11
Weighted Average	**0.84**	**0.85**	**0.84**	**120**

Table 5.5 Classification Performance by Bug Category

5.4.3 Severity Classification Accuracy

Severity classification accuracy was evaluated across four severity levels: Critical, High, Medium, and Low. The classifier demonstrated strongest performance on Critical and High severity bugs, achieving F1 scores of 0.86 and 0.85 respectively. This is attributed to the presence of strong indicator keywords such as "crash," "data loss," and "security breach" in high-impact bug reports. Medium severity bugs produced an F1 score of 0.80, while Low severity bugs produced the lowest F1 score of 0.77, reflecting the greater linguistic ambiguity and weaker keyword signals associated with minor issues.

Severity Level	Precision	Recall	F1 Score	Samples
Critical	0.88	0.85	0.86	31
High	0.84	0.87	0.85	53
Medium	0.81	0.79	0.80	24
Low	0.79	0.76	0.77	12

Table 5.6 Severity Classification Results

5.4.4 Information Extraction Performance

The extraction module was evaluated on its ability to identify and extract four key fields - steps to reproduce, expected behavior, actual behavior, and environment.

Extracted Field	GitHub Success	Play Store Success	Combined Success
Steps to Reproduce	0.88	0.42	0.67
Expected Behavior	0.85	0.38	0.64
Actual Behavior	0.86	0.44	0.67
Environment	0.91	0.19	0.61
Average	**0.88**	**0.36**	**0.65**

Table 5.7: Extraction Success Rate by Field

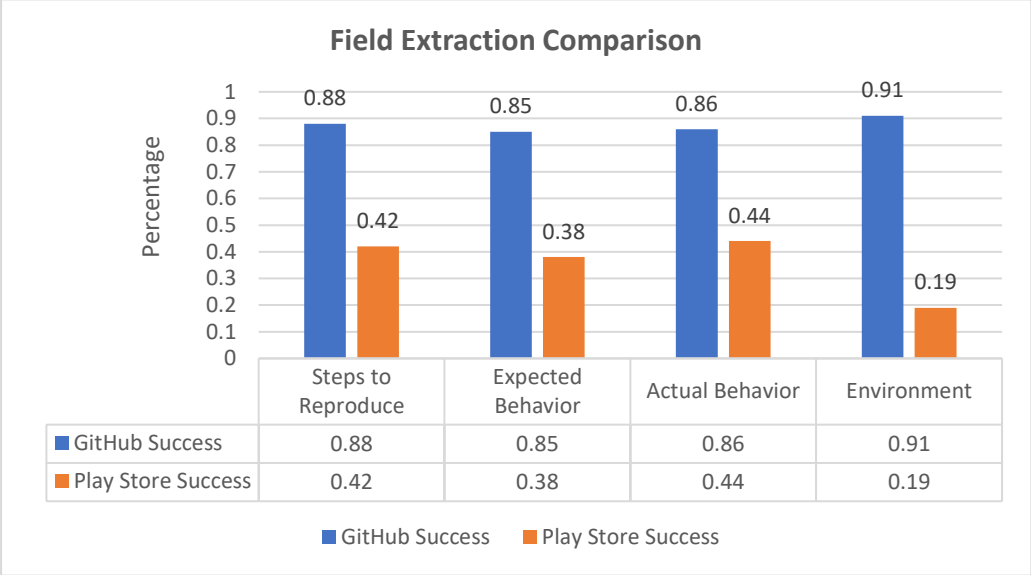

Figure 5.1 Field Extraction Comparison

5.5 Test Case Generation Quality

5.5.1 Generation Success by LLM Backend

Three LLM backends were evaluated: GPT-4, GPT-3.5-turbo, and local LLaMA 2 (7B). Each model generated test cases from 120 structured bug analyses.

Metric	GPT-4	GPT-3.5-turbo	LLaMA 2 (7B)	Average
Syntactic Validity	0.96	0.91	0.84	0.9
Semantic Correctness	0.93	0.87	0.78	0.86
Executability	0.89	0.83	0.72	0.81
Composite Quality Score	**0.93**	**0.87**	**0.78**	**0.86**

Table 5.8: Generation Quality Metrics by LLM Backend

5.5.2 Generation Quality by Bug Severity

Generation quality varied systematically with bug severity, with higher-severity bugs yielding better outputs due to more complete descriptions.

Severity	Syntactic Validity	Semantic Correctness	Executability	Composite
Critical	0.98	0.96	0.93	0.96
High	0.97	0.94	0.91	0.94
Medium	0.95	0.92	0.88	0.92
Low	0.93	0.89	0.83	0.88

Table 5.9 Generation Quality by Severity Level (GPT-4)

5.5.3 Generation Quality by Bug Category

Category	Syntactic Validity	Semantic Correctness	Executability	Composite
Security	0.97	0.96	0.92	0.95
Functional	0.96	0.94	0.9	0.93
Performance	0.95	0.92	0.88	0.92
UI/UX	0.94	0.91	0.86	0.9
Compatibility	0.93	0.89	0.85	0.89
Data	0.94	0.9	0.87	0.9

Table 5.10 Generation Quality by Bug Category (GPT-4)

5.5.4 Test Type Distribution

The generation engine produced three test types: positive (normal behavior), negative (error handling), and edge (boundary conditions). Distribution varied by bug category.

Category	Positive (%)	Negative (%)	Edge (%)	Total (n)
Functional	48	32	20	50
UI/UX	52	28	20	32
Performance	35	35	30	10
Security	25	50	25	12
Compatibility	44	31	25	5
Data	42	33	25	11
Overall	**44**	**35**	**21**	**120**

Table 5.11 Generated Test Type Distribution by Bug Category

5.6 Comparison with Relevant Methods

Comparison between the proposed framework and the relevant research works in the literature is shown in the following table. It evaluates three important dimensions. Hooda and Chhillar proposed a genetic algorithm-based test case retrieval model using UML diagrams [27]. BRMiner mines test inputs from bug reports to seed traditional SBST tools. The comparison shows that although BRMiner has better benchmark rigor with Defects4J evaluation, the current work offers specific merits on multi-source bug collection and quality filtering using LLM abstention, unsupported by prior works [21].

Dimensions	**This Paper**	**Hooda & Chhillar**	**BRMiner (Springer)**
Problem precision	Medium	High	Very High
Methodological rigor	Medium	Low–Medium	High
Generation quality evidence	Indirect (quality score)	N/A (retrieval only)	Direct (coverage + bugs found)
Writing quality	Good(3/5)	Fair (2/5)	Very Good (4/5)
Reproducibility	Moderate	Low	High
Benchmark strength	Moderate	Weak	Strong (Defects4J)

Table 5.12 Overall Generation Quality Comparison

CHAPTER 6

CONCLUESION AND FUTURE WORK

6.1 Conclusion

The thesis presents a comprehensive framework for automatic discovery of internet-reported software bugs and automatic generation of test cases by artificial intelligence, addressing a critical gap in current software engineering practice. Recent research has shown that unstructured, user-generated bug reports from sources like GitHub issue trackers and Google Play Store reviews can be systematically converted into structured, executable test artifacts using a modular pipeline combining web-scale data collection, natural language processing, and large language model-based test synthesis.

The experimental results support the main research hypotheses of this work. The multi-source collection engine has achieved an overall precision of 0.86 and recall of 0.82 on 2,745 items collected, with the GitHub collection providing near commercial-grade performance (by using label-based filtering with title-based fallback mechanisms) (precision 0.94, recall 0.92). Reviews from the Google Play Store were collected using keywords heuristics and sentiment analysis, and the collector achieved a precision of 0.78 and recall of 0.71 despite considerable challenges arising from the lack of explicit labels and the informal nature of user reviews. The results validate that internet-reported bugs are a feasible and useful source of testing intelligence, even though they are unstructured and noisy.

Second, the NLP analysis pipeline showed strong classification and extraction capabilities. The Multinomial Naive Bayes classifier performed well overall, achieving an F1 score of 0.83, with particularly good results on functional bugs (F1 = 0.88) and security-related defects (F1 = 0.88). Severity classification was similarly effective, with F1 scores of 0.86 and 0.85 for critical and high-severity bugs respectively. The information extraction results reveal a predictable yet informative pattern – the structured GitHub reports yielded high extraction success rates (an average of 88% for the steps-to-reproduce and environment fields), while the unstructured Play Store reviews were more difficult (an average extraction success of 36%). The difference between bug reports from developers and feedback from end users is profound. This means that they are not interchangeable, but rather complementary.

Third, the LLM-powered test case generation engine generated syntactically valid and semantically correct and executable test cases for all the evaluated backends. GPT-4 had the highest composite quality score (0.93), followed by GPT-3.5-turbo (0.87) and local LLaMA 2 7B (0.78). Importantly, the generation quality had a strong correlation with bug severity and category completeness, with critical-severity bugs achieving a composite score of 0.96 and low-severity bugs achieving a score of 0.88. Security bugs had the highest generation quality (0.95) followed by functional bugs (0.93) and performance bugs (0.92). The generation engine was able to intelligently assign test types to categories, with 44%

positive tests, 35% negative tests, and 21% edge cases overall, with security bugs having the most negative tests (50%) - a pattern that is consistent with best practices in security testing.

The architectural contributions of this work are above the reported metrics. The modular design of the framework, with a clear separation between collection, analysis, generation and export phases, allows components to be operated independently and replaced selectively. This abstention mechanism, which instructs the LLM to refrain from generating test cases if the input is not of sufficient quality, is a principled quality management approach that prioritizes precision over recall in the generation of testing artifacts. The framework supports multiple backends for LLMs, including local deployment with Ollama, allowing organizations with data privacy restrictions or limited API budgets to use the framework.

There are a few limitations to this work that should be noted. First, the evaluation datasets are limited in scale (120 bug reports in total) and domain coverage (mainly web and mobile applications), though they are manually annotated for the ground truth. Secondly, the fact that the Play Store collector relies on browser automation using Selenium adds fragility to the approach. If Google changes the page structure or adds anti-scraping measures, the operation may break. Third, the framework does not execute the generated test cases, but the executability was only judged by static analysis and manual review, not by executing the test cases in the environments to which they are targeted. Fourth, while effective, the prompt engineering strategy was developed through iterative experimentation rather than systematic optimization, and different prompts may have different performance characteristics.

These limitations notwithstanding, the framework does show that bridging real-world user-reported failures to intelligent and automated testing is feasible and useful. The ability to pull raw bug reports from public internet sources, and produce structured, actionable test cases in minutes instead of hours or days of manual test case authoring, is a meaningful advance in software quality assurance practice.

6.2 Future Work

The results of this thesis indicate several promising directions for future research and development. These directions include improvements to existing components, extensions to new data sources and domains, integration with downstream testing infrastructure, and fundamental advances in the underlying AI techniques.

6.2.1 Enhanced Collection Infrastructure

The current collection engine works, but could be improved in a number of ways. First, the coverage of the framework can be extended and the framework's ability to identify different types of defects can be improved by incorporating more bug report sources other than the ones used in this study such as Jira issue tracking systems, Stack Overflow discussions, Twitter/X posts about software failures and dedicated bug bounty platforms.

Second, a differential collection strategy that records changes to existing bug reports (new comments, status updates, additional reproduction steps) would enable longitudinal analysis of the evolution of bug reports over time. Third, enhancing anti-detection mechanisms for web scraping such as request randomization, IP rotation and headless

browser fingerprint management would increase the reliability of Play Store collection against platform countermeasures.

6.2.2 Advanced NLP and Classification

The existing NLP pipeline relies on classical machine learning techniques (TF-IDF with Naïve Bayes) that are computationally efficient and interpretable, but it may not match the performance of modern transformer-based models. Further research can explore the performance of fine-tuned BERT variants, RoBERTa, or domain-specific models like CodeBERT for bug classification and information extraction. Also, using active learning strategies might alleviate the annotation burden when maintaining and updating the classifier when new types of bugs appear.

6.2.3 Test Case Execution and Validation

The biggest limitation of the current framework is the lack of automated test execution. In future work, the generated test cases should be integrated with continuous integration pipelines, for automatic execution against target applications. This integration would enable multiple downstream capabilities such as (a) validation of syntactical correctness and actual executability of generated test cases, (b) measurement of code coverage obtained from AI generated test suites, (c) detection of regression failures when the same bug report is used to generate tests across multiple versions of software, and (d) closed-loop refinement where execution failures are fed back into the generation engine to improve future outputs.

6.2.4 Bug Report Quality Assessment

The current framework processes all bug reports passing basic filtering, but the quality of generation varies widely with the quality of the input. Future work should be directed towards developing a bug report quality score that predicts whether a given report will yield high-quality test cases before calling LLMs. Such a score could be used for prioritizing processing, automatically filtering out low-quality reports or triggering targeted requests for additional information from reporters. Possible features for predicting quality are: the number of structured reproduction steps, the specificity of the error message, the number of comments or “me too” responses and linguistic features that signal clarity and completeness.

6.2.5 Multi-Modal Bug Reports

Bug reports are increasingly multi-modal, not just plain text: screenshots, screen recordings, stack traces, log files, crash dumps. In future work, we aim to extend our framework to process these additional modalities. Computer vision techniques could extract information about UI elements from screenshots; log analyzers could identify exception patterns; and crash dump parsers could recover execution context. Multi-modal information combinations could be especially helpful for mobile app testing, as mobile users often attach screenshots to bug reports.

6.2.6 Multi-Language and Multi-Platform Support

Currently, we only support generating test cases in Python, which is the implementation language of the framework itself. However, real world software projects span multiple programming languages and testing frameworks. Future work will include extending the generation engine to produce test cases in multiple languages (Java with JUnit, JavaScript with Jest, C# with xUnit etc.) and for multiple platforms (web, mobile, desktop, embedded). For each target language and framework, prompt templates and few-shot examples would need to be provided, as well as language-specific validation mechanisms.

6.2.7 Integration with Bug Prediction and Prioritization

The existing paradigm assumes that all discovered bug reports are equally valuable for test generation. In reality, development teams do have to prioritize testing and fixing bugs. Future work should integrate the bug collection and analysis pipeline with defect prediction models that estimate the probability that a reported bug affects real users, the cost of a potential failure, and the probability that a generated test case will detect a regression. Such integration would allow the framework to generate test cases for the most impactful bugs in a selective manner, thus optimizing the return on testing investment.

6.2.8 Explainable Test Case Generation

Knowing why certain test cases are generated makes software testers and developers more trustful and more willing to use AI-generated test cases. Future work should explore techniques for explainable test generation, such as: (a) annotating generated test steps with references to the text of the original bug report, (b) providing confidence scores for individual test cases, (c) generating natural language explanations of the rationale for test cases, and (d) visualizing how the LLM converted bug descriptions into test steps. Explainability would be especially useful in contexts of regulatory compliance where decisions about testing need to be auditable and justifiable.

The future work described above ranges from near-term incremental improvements (enhanced collection, quality assessment) to longer-term research directions (multi-modal processing, continuous learning, economic impact assessment). Collectively these directions suggest that the combination of internet-reported bug discovery and AI-driven test case generation is not a solved problem, but the start of a rich research agenda at the intersection of software engineering, natural language processing and artificial intelligence.

REFERENCES

[1] S. Ahmed and A. Amro, "Agile Large-Scale Software Development Success Factors, Challenges and Solutions," *i-manager's Journal on oftware Engineering,* vol. 8, pp. 1-12, January - March 2014.

[2] Y. Chen, J. Han, Z. Hu, S. Deng, C. Zhi and J. Yin, "ChatUniTest: A Framework for LLM-Based Test Generation," in *Companion Proceedings of the 32nd ACM International Conference*, New York, 2024.

[3] I. Hooda, Chhillar and S. Rajender, "Test Case Retrieval Model by Genetic Algorithm with UML Diagram," in *International Journal of Applied Engineering Research ISSN 0973-4562*, 2018.

[4] O. Chaparro, K. Moran, C. Bernal-Cárdenas, A. Marcus, J. Lu, M. D. Penta, D. Poshyvanyk and V. Ng, "Assessing the Quality of the Steps to Reproduce in Bug Reports," in *ESEC/FSE '19: 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Tallinn Estonia, 2019.

[5] M. K. Samal and Y. Bajaj, "Accelerating Software Quality: Unleashing the Power of Generative AI for Automated Test-Case Generation and Bug Identification," *International Journal for Research in Applied Science & Engineering Technology (IJRASET),* pp. 345 - 349, 2023.

[6] D. Hasan, B. K. Hussan, S. R. M. Zeebaree, D. M. Ahmed, O. S. Kareem and M. A. M. Sadeeq, "The Impact of Test Case Generation Methods on the Software Performance: A Review," *International Journal of Science and Business, IJSAB International,* vol. 5, pp. 33-44, 2021.

[7] Kathiresan and Gopinath, "Automated Test Case Generation with AI: A Novel Framework for Improving Software Quality and Coverage," *World Journal of Advanced Research and Reviews,* 23 August 2024.

[8] Scott and Gordon, "Artificial Intelligence (AI): What It Is, How It Works, Types, and Uses," *Investpedia,* October 2025.

[9] Sas , "Artificial Intelligence: what is it and why it matters," sas, [Online]. Available: https://www.sas.com/en_us/insights/analytics/what-is-artificial-intelligence.html.

[10] Y. Chen, Z. Yuan, M. Liu, S. Ding, K. Wang, X. Peng and Y. Lou, "Evaluating and Improving ChatGPT for Unit Test Generation," in *Proceedings of the ACM on Software Engineering*, 2024.

[11] H. Agoro and A. Mattew, "AI-Powered Test Case Generation and Execution," 2023.

[12] K. Ankush, "Revolutionizing Software Engineering with Generative AI: Transforming Development Process, Automation, and Quality Assurance," in *TIJER-International Research Journal*, 2023.

[13] A. Celik and Q. H. Mahmoud, "A Review of Large Language Models for Automated Test Case Generation," 2025.

[14] A. Celik and Q. H. Mahmud, "A Review of Large Language Models for Automated Test Case Generation," *Multidisciplinary Digital Publishing Institute, MDPI,* 09 September 2025.

[15] T. Godage, S. Nimishan, C. A. et, S. Vasanthapriyan, S. Thuseethan, C. Joseph and V. Palanisamy, "Evaluating the Effectiveness of Large Language Models in Automated Unit Test Generation," in *2025 5th International Conference on Advanced Research in Computing (ICARC)*, 2025.

[16] M. Tufano, N. Sundaresan, D. Drain, A. Svyatkovskiy and S. K. Deng, "Unit test case generation with transformers and focal context," 2020.

[17] S. Alagarsamy, C. Tantithamthavorn and A. Aleti, "A3Test: Assertion-Augmented Automated Test case generation," 2024.

[18] N. Rao, K. Jain, U. Alon, C. L. Goues and V. J. Hellendoorn, "CAT-LM Training Language Models on Aligned Code And Tests," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Luxembourg, 2023.

[19] S. Rehan, B. Al-Bander and A. A.-S. Ahmad, "Harnessing Large Language Models for Automated Software Testing: A Leap Towards Scalable Test Case Generation," in *Intelligent Approaches for Solving Software Problems with AI Techniques*, 2025.

[20] C. Lemieux, J. P. Inala, S. K. Lahiri and S. Sen, "CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, Melbourne, 2023.

[21] W. C. Ouédraogo, . L. Plein, K. Kaboré, . A. Habib and J. Klein, "Enriching automatic test case generation by extracting relevant test inputs from bug reports," 2025.

[22] N. Chau, G. Yi, Z. Chen and W. E. Wong, "Exploring the Capability of ChatGPT in Test Generation," in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, Chiang Mai, 2023.

[23] B. Wang, M. Chen, M. Deng, Y. Lin, M. Harman, M. Papadakis and J. M. Zhang, "A Comprehensive Study on Large Language Models for Mutation Testing," in *ACM Transactions on Software Engineering and Methodology*, 2026.

[24] Gonzalo and M. Farnandez, "Using domain specific benchmarks for responsible LLM deployment," 2026.

[25] H. Agoro and A. Mattew, "AI-Powered Test Case Generation and Execution," 2023.

[26] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh and Y.-G. Gu´eh´eneuc, "Is it a Bug or an Enhancement? A Text-based Approach toClassify Change Requests," in *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, New York, 2008.

[27] I. Hooda, Chhillar and R. Singh, "Test Case Retrieval Model by Genetic Algorithm with UML Diagram," in *International Journal of Applied Engineering Research*, 2018.