A Unified Framework for Verification & Improvement of LLM-Based Automated Unit Test Generation

Muhammad Abbas (BSE223183)

Syed Mukhtar-ul-Hassan (BSE223208)

Muhammad Hammad Ali (BSE223005)

Supervised By

Mr. Mudassar Adeel Ahmad

Fall 2025

Design Project-I

Department of Software Engineering

Capital University of Science & Technology, Islamabad

Project Proposal

VERSION	V X.0

NUMBER OF MEMBERS	03

TITLE	"A Unified Framework for Verification & Improvement of LLM-Based Automated Unit Test Generation."

SUPERVISOR NAME	**Mr. Mudassar Adeel Ahmad**

MEMBER NAME	REG. NO.	EMAIL ADDRESS
Muhammad Abbas	BSE223183	abbas63891@gmail.com
Syed Mukhtar-ul-Hassan	BSE223208	shahmukhtar106@gmail.com
Muhammad Hammad Ali	BSE223005	hammadmuhammad2288@gmail.com

MEMBERS' SIGNATURES

_____

_____

_____

Supervisor's Signature

Approval Certificate

This project, entitled as “A Unified Framework for Verification & Improvement of LLM-Based Automated Unit Test Generation” has been approved for the award of

Bachelors of Science in Software Engineering

Committee Signatures:

Supervisor: _____

(Supervisor Name Dr / Mr / Ms)

Project Coordinator: _____

(Mr. Ibrar Arshad)

Head of Department: _____

(Dr. Nadeem Anjum)

Declaration

We hereby declare that the work presented in this project, "A Unified Framework for Verification & Improvement of LLM-Based Automated Unit Test Generation", is the result of our original effort and has not been submitted, either in whole or in part, for any other degree or qualification at this or any other institution. All references to existing work, whether published or unpublished, have been duly acknowledged and cited to give proper credit to the original authors.

Furthermore, we affirm that this project complies with the ethical and academic standards established by Capital University of Science & Technology.

MEMBERS' SIGNATURES

_____

_____

_____

Acknowledgements

We would like to express our sincere gratitude to Mr.Mudassar Adeel Ahamd, Our Project Supervisor, for his guidance, encouragement and continuous support. His insights and constructive feedback played a crucial role in shaping our work and ensuring it.

We also extend our appreciation to Our University, for providing us with the necessary resources and a conductive environment to begin and complete our Final Year Project.

Table of Contents

List of Figures

LIST OF TABLES

This page is kept blank

Chapter 1

1. Introduction

Software testing is a process of making sure that software functions as it is expected to before it is implemented, and there are a number of levels involved in software testing, including unit testing, the integration testing and the acceptance testing. Integration testing is used to ensure the interaction of modules, and acceptance testing is used to determine whether the system complies with requirements of business and users [16][17]. In spite of the fact that both integration and acceptance testing have grown up to be fully automated with application frameworks such as Selenium and Robot Framework [18], unit tests continue to be highly manualized and a development bottleneck [1].

A method of minimizing this effort is automated generation of unit tests by investigators with tools like EvoSuite and Randoop, which tend to generate unreadable or trivial tests [2]. Large Language Models (LLM) such as GPT-3.5 (OpenAI) [19], CodeT5+ (Salesforce) [20], StarCoder (Hugging Face) [21], and Code Llama (Meta AI) [22] have provided the opportunity to generate readable and seemingly useful tests. Nevertheless, the limitations to this project are that, the existing LLM-based frameworks are still plagued by such problems as poor handling of external dependencies, low repair effectiveness, and weak assertions.

In spite of this promise, major challenges to test generation based on LLM still remain. Most of the generated tests do not run because of external dependencies are missing [1], include weak or trivial assertions that are not adequate to reflect faults in reality [6], and exhibited limited success in repair when adaptive loops are used [7]. Even though the individual problems of the research have been solved in previous studies, e.g., by enhancing assertions [5], by mutation testing [3], or by minimizing duplicate tests, the current solutions are still scattered. The proposed project is a unified framework based on verification and improvement. Along with mocking support of missing external dependencies, repair loops, and a user-

friendly interface, framework aims to make unit tests more practical and better than those being used now in figure 1-1.

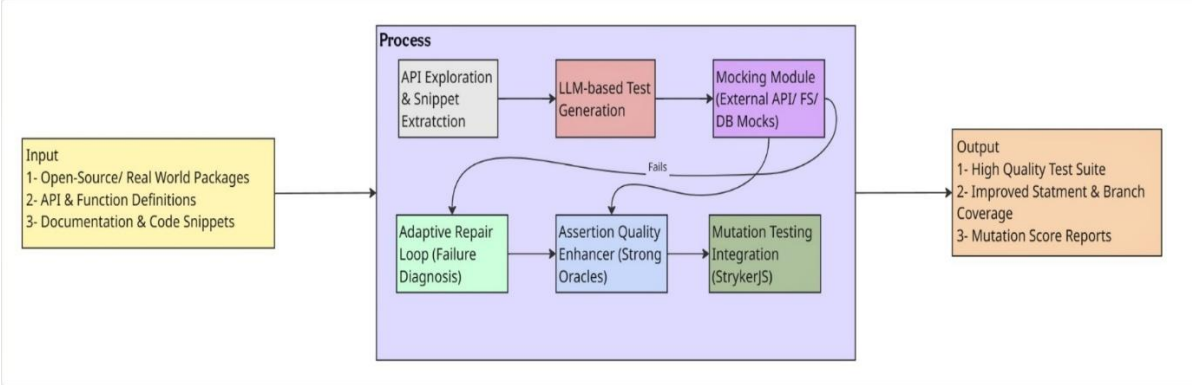

Figure 1-1 Overview Diagram

1.1. Project Introduction

The purpose of the proposed project is to create a workable system to verify and enhance large language model (LLM) generated unit tests. It aims to fill the gap between the automated test generation and real-world software quality assurance with the help of the practical testing and the process of the step-by-step improvements. The major components that are integrated into the system include the following:

- **Automated detection and mocking of external dependencies**: Discovers APIs, databases or file system calls and adds the right mocks or stubs automatically.
- **Multi-step adaptive repair loop with structured error diagnosis**: Examines test failures and controls the LLM to a process of informed repair.
- **Assertion-quality enhancer**: Detects weak or trivial assertions and strengthens them using semantic understanding and runtime behavior.
- **Developer-oriented interface**: Provides a user-friendly environment, such as a VS Code extension or web dashboard, enabling developers to review, validate, and refine generated tests interactively.

The framework aims to turn weak and automatically generated tests into stronger, simpler, and more useful test suites that developers can easily use in real projects while contributing to the research on integrating verification and improvement in LLM-based software testing..

1.1.1. Problem Statement

LLM-generated unit tests produce readable and apparently useful tests but are frequently unexecutable, weak at detecting real faults, and impractical for developer workflows. They break on code that uses external resources because automatic mocking is missing, repair loops fix only a small fraction of failing tests, assertions are often trivial, mutation-based fault detection is low, and existing tools lack an integrated developer-friendly interface.

There is a need for a unified framework that verifies LLM-generated tests to make them executable and fault-sensitive, and which simultaneously improves them through stronger assertions, and targeted test generation, while providing a seamless development environment for practical developer adoption.

1.1.2. Context and motivation

Recent work demonstrates that LLMs can produce unit tests and often outperform traditional automated test generators on coverage metrics; however, important limitations remain. The initial work on LLM-based test generation (LLM TESTPILOT) [1] demonstrated promising statement/branch coverage improvement, but also demonstrated the same weaknesses: a large portion of generated tests fail in practice because they lack mocks, the degree of improvement in test generation is substantial but not as high as it has been noted with human-written suites [1], the assertions are trivial or weak, generation of tests via mutation continues to be lower than with human-written suites [1]. Independently, research on mutation testing [2], assertion generation[3], LLM-driven mutant creation, and JS-specific test generation provides partial solutions but typically addresses only one aspect (verification or improvement) [4], leaving a clear opportunity for a unified pipeline that combines both verification and improvement steps [5][6][7].

1.1.3. Scope and constraints

The project will focus initially on JavaScript / Node.js (the same ecosystem evaluated in the baseline study) because dynamically typed languages present high-value challenges such as type inference, and environment/dependency mocking [2]. Implementation will integrate existing tools where feasible (e.g., StrykerJS for mutation testing, Sinon/Nock for mocks) while extending them with LLM-guided generation and repair strategies. The evaluation will measure both traditional coverage metrics and mutation scores to show improved fault-detection power, and will include developer-centric usability checks via the proposed VS Code extension [9].

1.2. Existing Examples / Solutions

Historical unit test generation tools based their generation on search-based and symbolic execution algorithms, including EvoSuite with Java [2] and Randoop with general-purpose languages. These systems are usually trying to achieve the highest code coverage by producing vast spaces of inputs by using evolutionary search or random sampling of program paths. Although they were applicable in the controlled experimental setting, their application in actual projects was minimal. These tests generated were usually hard to maintain, much manual effort was needed to configure them to the environment and most importantly, there were no strong oracles to test the correctness of the program[14]. It was subsequently replaced by mutation testing as a stricter measure to assess fault-detection capability which made generated tests more realistic [3]. But to scale mutation analysis to the level of an industrial project, it needed vast computing capacity and a lot of engineering support as is evidenced in the large scale use of mutation testing in production by Google [2].

Table 1-1 Comparative Analysis oF Related Research

#	Reference	Input	Verification	Improvement	Output
1	Tip et al. (2025)	JavaScript code	✓	✗	Mutants
2	Primbs et al. (2025)	Java Existing Tests + focal methods	✗	✓	Assertions
3	Nan et al. (2025)	Java code + test intentions	✓	✗	Unit test suite
4	Wang et al. (2025)	Project-level code (multi-language)	✓	✗	Unit test suite
5	Schäfer et al. (2024)	JavaScript code	✓	✗	Unit test suite
6	Sánchez et al. (2024)	OSS developers & projects (multi-language)	✗	✓	Developer insights
7	Alagarsamy et al. (2024)	Java code	✗	✓	Unit test suite
8	Mali et al. (2024)	Hardware designs / specs	✓	✗	Assertions
9	Olsthoorn et al. (2024)	Server-side JavaScript codebases	✓	✗	Unit test suite
10	Chen et al. (2024)	Java code snippets + context	✓	✗	Unit test suite
11	Yang et al. (2024)	Java projects	✗	✓	Evaluation metrics
12	Petrović et al. (2022)	Large-scale codebases (multi-language)	✓	✗	Mutation testing report
13	Sánchez et al. (2022)	Open-source repositories (multi-language)	✓	✗	Adoption dataset
14	Stallenberg et al. (2022)	JavaScript code (no static types)	✓	✗	Test cases
15	Park et al. (2021)	JavaScript programs / spec	✓	✗	Test programs
16	Our Proposed Solution	JavaScript code	✓	✓	Verified, Improved Test Suite

1.2.1. Identified shortcomings in existing solutions

Based on the above survey, it can be noted that the available systems are deficient in three primary areas. To start with, they are usually very narrow in their focus and usually focus on either the assertion quality, mocking, or mutation analysis analysis, but hardly on a combination of several aspects in the same pipeline. Second, the actual verification is poor, with majority of studies focusing on coverage measures and disregarding strict measures like mutation tests in feedback loops. Third, enhancement is severely neglected, which leads to the presence of redundant, needless, or insignificant suites.

The given framework is aimed at addressing these shortcomings and unifying verification and improvement. This project will provide test suites which, in addition to being executable, are effective, short, and usable in real-life scenarios through the combination of mutation testing, test repair, assertion strengthening and minimization.

1.3. Business Scope

Business and innovation wise, the framework proposed has a great potential in the emerging subject of AI-assisted engineering of software. With the growing use of large language models (LLMs) in organizations to generate code and perform automation, the need to have **verifiable, maintainable**, and **trustworthy test suites** is on the increase. This project addresses a major gap in the market by offering an integrated solution that combines **verification (mutation testing, repair loops)** and **improvement (assertion strengthening)** into a single developer-friendly platform.

Table 1-2 Lean Canvas

Problem	**Solution**	**Unique Value Proposition**	**Unfair Advantage**	**Customer Segments**
1. LLM-generated tests fail due to missing mocks/external dependencies. 2. Assertions are weak and trivial, lowering bug-detection power. 3. Repair loops fix <15% failures and lack effectiveness. 4. Redundant or bloated tests waste developer time. 5. Current tools are CLI-based	A unified framework that: 1. Verifies tests via mutation coverage 2. Improves assertions automatically 3. Repairs tests using adaptive loops 4. Use mocking to handle external dependencies for reliable test execution. 5. Provides an interactive VS Code/web dashboard	"The first unified framework that not only generates but also **verifies and improves** LLM-based test suites, ensuring they are executable, effective, and practical for developers."	Integration of both verification and improvement in one pipeline (rare in literature). Combination of mutation testing, oracle enhancement, repair, and GUI-based interface gives it a competitive edge.	Software development companies AI/ML-driven testing tool vendors Freelance developers Academic researchers in SE/AI Open-source maintainers
Existing Alternatives	**Key Metrics**	**High-Level Concept**	**Channels**	**Early Adopters**
Existing systems: EvoSuite, Randoop, ChatUniTest, SynTest-JS, A3Test. Their shortcomings: weak assertions, poor repair loops, no integration of mutation testing, lack of	Mutation score improvement (vs. baseline). Assertion quality metrics (strong vs trivial). Test suite reduction ratio. Developer adoption (via plugin installs).	"GitHub Copilot for testing but smarter, because it doesn't just generate tests, it verifies and improves them."	Open-source distribution (GitHub, VS Code Marketplace). Integration with CI/CD platforms. Academic conferences/journal.	Research labs working on LLM-based SE. Early-stage startups building AI coding assistants. Open-source contributors in LLM-generated code/tests.

developer-friendly UI.				
Cost Structure			**Revenue Structure**	
Development costs (framework design, VS Code extension). Hosting costs for dashboards/analysis. Compute resources for LLM inference and mutation testing. Maintenance and open-source community support.			Open-source core + premium enterprise version. SaaS subscription for automated verification dashboards. Enterprise licensing for large-scale CI/CD integration. Consulting and customization services for companies adopting LLM-based testing.	

1.4. Useful Tools and Technologies

The successful design and implementation of this project requires selecting appropriate tools and technologies that support mocking, automated repair, assertion enhancement, mutation testing, and developer-friendly integration. The following subsections describe the chosen technologies and their justifications.

Table 1-3 Tools and Technologies Used in the Project

Category	Technology / Tool	Technical Justification
Programming Language	JavaScript/ TypeScript	Selected due to native support in the base framework (*TestPilot*) [1] and mutation testing tool (*StrykerJS*). Ensures compatibility with Node.js projects and npm packages, reflecting real-world JavaScript testing environments.

Development Environment	Visual Studio Code (VS Code), Node.js & npm, Git & GitHub	VS Code provides strong ecosystem support for JavaScript/TypeScript, testing frameworks, and custom extension building. Node.js offers runtime execution and package management, while Git & GitHub support collaborative version control and open-source workflow.
LLM APIs and Models	OpenAI GPT-3.5 Turbo	Serves as the primary LLM for unit test generation, repair, and assertion enhancement. Focuses on improving GPT-based test reliability through verification and adaptive repair mechanisms.
Testing and Verification Tools	StrykerJS, Mocha / Jest, Sinon / Nock	StrykerJS performs mutation testing to assess and improve fault-detection capability. Mocha and Jest are used to execute and validate generated test cases, while Sinon and Nock handle mocking and stubbing of external dependencies such as APIs and file systems.
Supported Platforms	Windows, Linux, macOS	Ensures full cross-platform compatibility since the system is implemented using Node.js and developed within VS Code.

1.5. Project Work Break Down

The project will be designed with a work-based Work Breakdown Structure (WBS) that will be structured based on literature review and framework design to implementation, evaluation, and documentation that are divided into different stages. Every stage is an embodiment of the technical and research efforts of the team, as the workload was evenly distributed, and the goals of the base framework were met (TestPilot). All the three members (Abbas, Mukhtar and Hammad) share responsibilities as they make sure that they collaborate, specialize, and make consistent progress throughout the project lifecycle as indicated in figure 1-2.

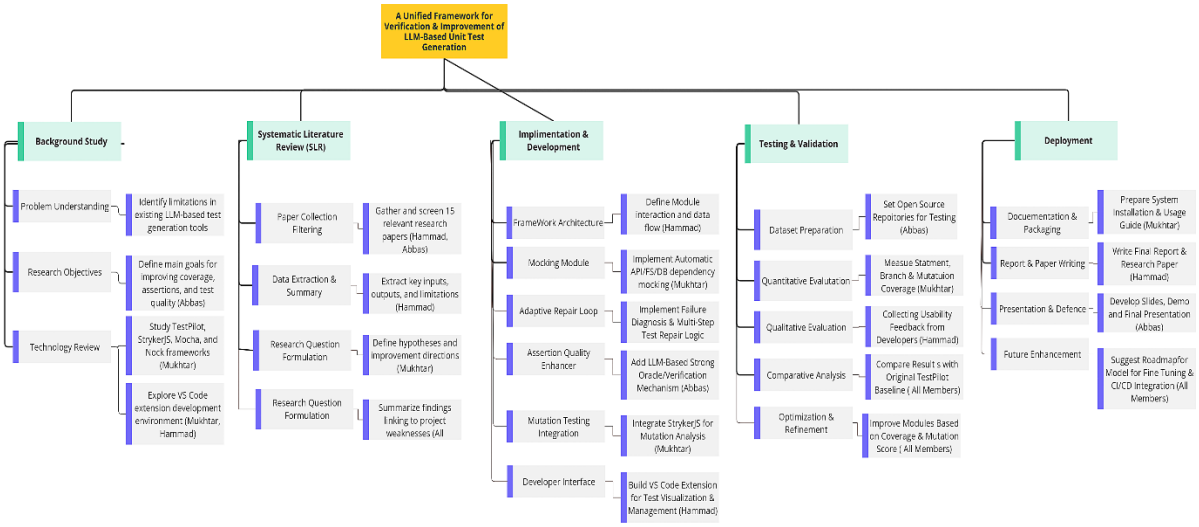

Figure 1-2 Work Breakdown Structure

1.6. Project Time Line

The project is set to occur within the period of **October 2025 to June 2026** covering research, design, implementation, evaluation, and documentation. The schedule will start with the identification of the problem, literature review, and framework design, and then with a gradual implementation which will be based on the use of mocking, adaptive repair, assertion enhancement, and mutation testing. The next phases are empirical testing on the TestPilot framework, its performance tuning and refinement by users, and its documentation and defense. All the duties are shared between the team members: Abbas, Mukhtar, and Hammad in order to balance the progress and consistent development of the project lifecycle as illustrated in figure 1-3.

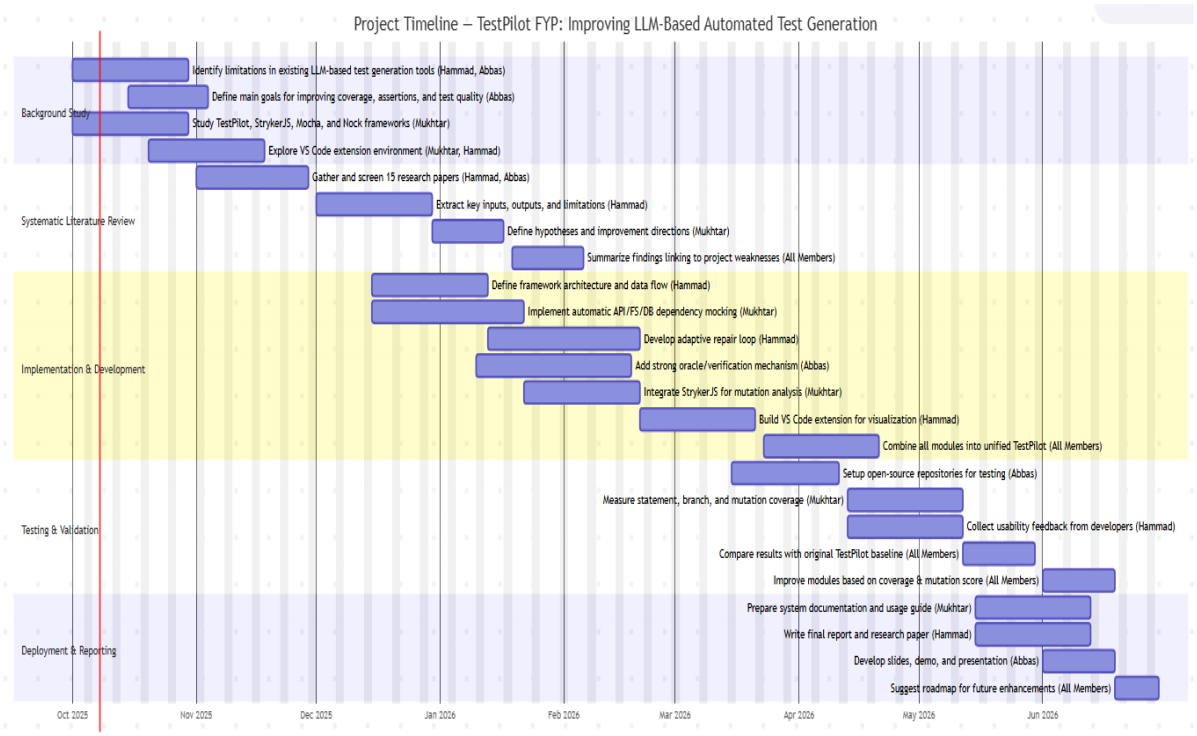

Figure 1-3 Gantt Chart

Chapter 2

2. Background Study

This chapter offers a whole background to the field of research by analyzing the current literature, automated test-generation methods that have already been used, and the current developments in NLP and Large Language Model (LLM) based testing. It starts with an extensive literature review that summarizes the previous works, classifies their approaches and describes the present state of automated unit test generation. The chapter further compares the research findings of the past, noting the essential differences between the conventional methods of testing, contemporary tools of the LLM and verification methods of the chosen studies. Lastly, the gaps and limitations in the current literature are identified to come up with the objectives of the research that will be used to inform the direction of this thesis.

2.1 Literature Review

This literature review aims to study the literature that is associated with automated unit test generation, both conventional and current using the LLM. This section is a synthesis of the results of the chosen 48 primary works, a description of the methods and techniques employed in the previous study, and the evolution of the testing structures, algorithm, and verification plans. The systematic review of these studies would help to form the background knowledge needed to comprehend the existing developments, pinpoint existing shortcomings, and situate the contribution made by this thesis in the context of research in general.

2.1.1 Introduction

Software testing is an early stage of software development lifecycle and it is a necessary element to software reliability, correctness and maintainability. Of the many levels of testing, unit, integration, system, and acceptance, unit testing is most significant as it only confirms

individual program modules to others before intermingling with the rest of the system. Although this is a vital process, the creation of unit tests is still a very labor-intensive and time-consuming process that involves human-engineered creation of meaningful inputs, expected outputs and assertions, which are subject to human error. The traditional methods of automated test generation, including search-based, evolutionary, or static-analysis-based methods, have tried to relax this burden, but still remain problematic in generating strong oracles, dealing with external dependencies, and keeping executable test in practice (Abdallah et al., 2018; Olsthoorn et al., 2024).

The development of Natural Language Processing (NLP) and Large Language Models (LLMs) has opened up new opportunities related to automation of software engineering, especially in the work with program understanding, code generation, and test generation. Modern LLMs like GPT-series and CodeT5+ as well as StarCoder and Code Llama are able to extract semantic code information, derive meaning from natural language descriptions, and generate human readable tests that better resemble tests written by developers when compared to previous automated methods (Schaefer et al., 2024; Zhang et al., 2025; Li et al., 2024). Such models are able to comprehend both documentation and source code and this is enabled through NLP techniques, including semantic parsing, intent extraction, and text-to-code translation that allows more expressive and context-aware test generation (Taromirad et al., 2025). However, irrespective of these developments, tools produced by LLM are usually susceptible to more realistic constraints, such as the absence of mocks-to-external-dependency, trivial or weak assertions, compilation failures, hallucinated behaviour and failure to detect faults when tried against more demanding testing standards, such as mutation testing (Petrović et al., 2022; Sánchez et al., 2022; Tip et al., 2025).

Conventional non-LLM-based test generation systems also keep developing alongside them. The tools that are based on these techniques, like EvoSuite, Randoop, Nessie, GuessWhat, SynTest-JS, and PC-TRT, support the effectiveness with a wide range of programming languages and execution models (Stallenberg et al., 2022; Kim et al., 2022; Nguyen et al., 2022). Although these classical tools are effective in covered situations, empirical evidence has continuously revealed that they are unable to cope with oracle quality, dependency mocking, and scaling to the complex industrial projects (Li et al., 2018). Similarly, the tools

that target the particular weakness (e.g., A3Test, ChIRAAG) or the test amplification (e.g., Small-Amp) focus on particular weaknesses but do not offer the comprehensive solutions that combine the processes of generation, validation, and improvement in a single pipeline (Abdi et al., 2022; Alagarsamy et al., 2024; Mali et al., 2024).

Both the frameworks based on LLM and non-LLM-based still face verification and quality improvement as a persistent challenge. Previous research indicates that a high percentage of automatically generated tests are undiscoverable, flaky, redundant, or incomplete, and more verification steps (such as mock generation, assertion strengthening, mutation analysis, adaptive repair loops, or minimization) are needed to ensure that tests are correct and practically useful (Nan et al., 2025; Chen et al., 2024; Gu et al., 2024). Specifically, mutation testing has become a focal point of assessing the effectiveness of tests in general and quantifying the ability to detect faults in particular. Nevertheless, the scale of mutation testing adds computational complexity, which further explains the necessity of effective verification principles and combined remediation plans (Harman et al., 2025; Petrović et al., 2022).

Although there is a large amount of literature discussing the various test generation methods, the literature is still disjointed. The current surveys are more likely to concentrate on particular themes e.g., assertion generation, mutation testing, NLP-assisted testing, or the workflow of development powered by LLMs without offering a single perspective to link NLP methods, architecture of tools, algorithmic basis, and verification tactics with gaps in the field. Additionally, the acceleration of the development of LLM-based tools since the years 2021 (Jiang et al., 2024) also indicates the necessity of a systematic review to synthesize the developing situation, evaluate the current opportunities and discover research opportunities.

To fulfill this requirement, this paper performs a Systematic Literature Review (SLR) of automated unit test generation across both those based on LLM and those not based on LLM. The review is based on the accepted guidelines of SLR and examines peer-reviewed articles published between 2018 and 2025 and found in large digital repositories, such as IEEE Xplore, ACM Digital Library, SpringerLink, and Elsevier. The review is organized into five research questions which together address NLP techniques, methods of verification, tools, algorithms, and open research issues:

- **RQ1:** What Natural Language Processing (NLP) techniques have been applied for automated test generation and software testing?
- **RQ2:** How do existing frameworks verify or optimize the quality of generated tests?
- **RQ3:** What tools have been developed and/or used for automated unit test generation, and/or what input do these tools take and what output do they produce?
- **RQ4:** What algorithms have been developed and/or applied in automated unit test generation, how do these algorithms transform system inputs into test outputs?

These questions will help this review, therefore, present a cohesive picture of the present state of automated unit test generation, identify the major shortcomings of the current methods, and show the promising future of the research. The results of this review will be of use to researchers, tool developers, and practitioners who want to develop more powerful, provable, and industry-capable test generation frameworks.

2.1.2 Research Methodology

In this section, the researcher describes the methodological steps that will be used in the process of conducting this Systematic Literature Review (SLR). The methodology was formulated according to the available guidelines on SLR to achieve rigor, reproducibility and transparency during the review process. It establishes the classification categories that were derived to be analyzed, the review protocol and how the studies were selected, evaluated, and synthesized.

2.1.2.1 Category Definitions

Answering the research questions, based on the preliminary proposal, and the themes identified in the previous literature, five general categories were obtained to structure the extracted data (Schaefer et al., 2024; Li et al., 2024; Sanchez et al., 2022). These typologies make it easier to analyse automated unit test generation in both the LLM-based and non-LLM-based perspectives.

- **Category 1: NLP Techniques for Test Generation**

This category classifies Natural Language Processing (NLP) methods that are applied in the generation of tests, understanding code semantics, deriving assertions or requirements. Such techniques are semantic parsing, intent extraction, information retrieval, and text-to-code transformation (Taromirad et al., 2025; Nguyen et al., 2022).
Addresses **RQ1**.

- **Category 2: Verification and Improvement Approaches**
  This category comprises the methods of demonstrating the validity and effectiveness of generated tests. They can be mutation testing, dependency mocking, assertion quality testing, adaptive repair loop, and code execution testing (Petrović et al., 2022; Gu et al., 2024).
  Addresses **RQ2**.
- **Category 3: Tools and Frameworks for Automated Test Generation**
  New tools are suggested in studies or old frameworks reused (e.g. SynTest-JS, EvoSuite, ChatUniTest, A3Test, TestPilot, Nessie). Each tool has its category of input (e.g., source code, test intentions) and output (e.g., unit tests, assertions, mutants) (Olsthoorn et al., 2024; Chen et al., 2024).
  Addresses **RQ3**.
- **Category 4: Algorithms and Transformation Techniques**
  The category is concerned with algorithms that advise automated test generation, including search-based optimization, probabilistic type inference, neural generation, reinforcement learning-based repair, and method slicing (Stallenberg et al., 2022; Wang et al., 2025; Mani et al., 2025).
  Addresses **RQ4**.

These four categories are in direct correlation to the RQs and guarantee that such extracted data can be mapped, compared, and generalized systematically.

2.1.2.2 Review Protocol Development

In order to make the SLR a structured, unbiased and replicable process, a formal review protocol was set before the process. The protocol is composed of four main elements namely selection and rejection criteria, search process, quality assessment and data extraction with synthesis.

Table 2-1 Details of Search Terms with Operators and Search Results

#	Search Term	Operator	IEEE	Springer	Elsevier	ACM
1	Large Language Model	N–A	40,331	3,676,024	785,517	311,322
2	Automated Test Generation	N–A	9,149	187,152	289,945	138,227
3	Unit Test Generation	N–A	19,793	467,785	1,000,000+	196,590
4	Mutation Testing	N–A	8,688	218,495	1,000,000+	22,344
5	Assertion Generation	N–A	588	81,526	62,531	51,259
6	Test Verification	N–A	40,669	351,867	1,000,000+	281,869
7	JavaScript Testing	N–A	2,710	18,011	13,278	16,831
8	LLM-based Testing	N–A	436	9,214	18,511	12,131
9	ChatGPT	N–A	3,234	21,506	36,449	8,479
10	“Large Language Model” AND “Unit Test Generation”	AND	330	218,687	14	105,963
11	“ChatGPT” AND “Automated Testing”	AND	141	3,629	119	4,182
12	“Test Verification” OR “Test Optimization”	OR	159,247	44,777	_	409,463
13	“Mutation Testing” AND “LLM”	AND	56	602	37	1,084
14	“Fault Detection” AND “Unit Testing”	AND	3,250	22,936	381	104,608
15	“Assertion Generation” AND “ChatGPT”	AND	7	329	5	1,781
16	“Test Oracle” AND “Automation”	AND	241	4,882	292	10,164
17	“JavaScript” AND “Test Generation”	AND	417	9,087	99	16,539
18	“Dynamic Languages” AND “LLM-based Testing”	AND	50	1,164	1	7,150
19	“Adaptive Loop” OR “Self-Healing Tests”	OR	39,214	88,580	2,087	124,044
20	“Mutation Coverage” AND “Automated Unit Test”	AND	97	4,229	1	4,030

i) Selection and Rejection Criterion

They establish definite rules of inclusion and exclusion of research works. The scope of this review is guided by five research questions and five pre-defined categories. In order to derive the right and credible answers to our research questions, we created six parameters. The parameters to be satisfied by the research works chosen will be in the following manner:

1. **Subject-relevant:**

   Only research works that are directly applicable to automated unit test generation, software testing, NLP-based test automation, or LLM-driven testing are worth selecting. The research has to help respond at least one of the five questions in the research and it has to fall into either of the five preset categories.

   Eliminate research works which are not within the area of automated test generation, or which do not correspond to any specified category.

2. **2018-2025:**

   The chosen research works should be published within 2018-2025. This era reflects the dawn and the steep rise of the LLM-aided test generation which has been affirmed by recent surveys which reveal that the after 2018, the scale of the LLM and NLP-based test-generation studies is on a significant upward steep rise (Jiang et al., 2024).

   exclude all studies that have been published prior to 2018 to be able to consider current and pertinent developments.

3. **Publisher:**

   The chosen research efforts should have been published in either of the four well-known scientific databases namely, IEEE, SPRINGER, ELSEVIER, and ACM. These are internationally acknowledged publishers with high-quality requirements and rigorous peer review in addition to authoritative contributions to the research of software engineering and AI (ACM Books, 2024; IEEE Author Center, 2024).

   Disregard the research that was published beyond these databases to preserve trust and research authority.

4. **Crucial-effects:**

   The research works to be selected should have meaningful contributions or effective findings in regard to automated unit test generation. The contributions can be in the form of new algorithms, tools, verification procedures, structures or empirical assessments illustrating the enhancement of test quality, assertion strength, mutation ratings or test runnability.

   Disregard studies which fail to offer any meaningful implications or information that can be applied to the context of automated test generation or to testing through LLM.

5. **Results-oriented:**

   Selected research works must be supported by solid validation, such as empirical evaluation, benchmarking, case studies, user studies, or controlled experiments. Studies must demonstrate reproducible results, clear methodology, and measurable outcomes. Reject studies with weak validation, insufficient experimentation, or anecdotal evidence.

6. **Repetition:**

   In cases where multiple publications cover essentially the same research context (e.g., extended versions, companion papers, or duplicated datasets), only the most comprehensive or most recent work will be selected.

   Reject repeated or redundant studies to avoid duplication and maintain diversity in the dataset.

ii) Search Process

The criteria of selection and rejection, prove that the four scientific databases **IEEE, ELSEVIER, SPRINGER,** and **ACM** will be used to carry out this SLR. These scientific databases issue high-impact journals, conferences, and workshops in software engineering, artificial intelligence, and computational linguistics, which, in its turn, makes them very relevant to our research environment. Besides the search of databases, the most pertinent survey papers, technical reports, and foundational books were also consulted to facilitate key concepts and confirm the trends mentioned in the studies analyzed.

In order to achieve the search process, they applied a combination of broad and narrow search terms based on our research questions. These were the words that related to the automated unit test generation, LLM-based testing, NLP techniques, assertion generation, mutation testing, and the verification frameworks. The following examples of the search terms are presented: Automated Test Generation, Large Language Model, Unit Test Generation, Assertion Generation, Mutation Testing, LLM-based Testing, JavaScript Test Generation, Dynamic Languages and Test Verification. **Table 2-1** presents the search terms and the acquired results of each scientific database.

All search terms had a year filter of **2018-2025** so that all the recent studies would be included due to the fast emergence of the trend in the research of LLM-based software engineering. A few of the search terms included a single keyword (e.g. ChatGPT) where logical operators (AND/OR) could not be used hence are highlighted in **Table 2-1** with the "N-A" mark. Although the "AND" operator was used to reduce the number of search combinations to very narrow ones, this did not necessarily mean that it was relevant to automated unit test generation. Because of this reason, the "OR" operator was also used to approximate the wider potential results. Nevertheless, the "OR" operator will inherently produce thousands of results, which is impossible to inspect manually. Consequently, search filters that were applied in all databases like advanced search filters like search within title, search within abstract, search within keywords and subject area filters were used to ensure that it is precise and relevant.

In the same vein, further specific or derivative search terms were added to the search space after the initial results were manually inspected to limit the search space to niche concepts. Examples are test oracle generation, probabilistic type inference and mutation-guided test generation. Such refinements made sure our search included the general approaches and also the highly specific frameworks as far as the research questions were concerned. There were screenshots and raw search logs that were stored as supplementary evidence in regards to traceability in relation to **Table 2-1**.

Table 2-2 Data Extraction and Synthesis Details for Selected Researches

#	Data Extraction / Synthesis Element	Description of Extracted Information
1	Research Identification	Title, authors, year of publication, type of publication (journal, conference, workshop), and database (IEEE, ACM, Springer, Elsevier).
2	Relevance to Research Questions	Identification of whether the study addresses one or more of the five research questions (RQ1–RQ5).
3	NLP / LLM Techniques Used	Extraction of NLP techniques (e.g., semantic parsing, intent extraction, IR-based methods) and LLM capabilities (e.g., GPT-4, CodeT5+, StarCoder).
4	Verification and improvement Methods	Extraction of validation strategies used in the study, including mutation testing, assertion enhancement, repair loops, mocking, and coverage evaluation.
5	Tool / Framework Identification	Identification of developed tools or reused frameworks, including tool inputs (e.g., source code, test intentions) and outputs (e.g., unit tests, assertions, mutants).
6	Algorithmic Approach	Extraction of core algorithm(s) used (e.g., evolutionary search, probabilistic type inference, RL-based healing, method slicing).
7	Category Mapping	Mapping of each research work to one or more of the five predefined categories.
8	Key Contribution Analysis	Identification of the primary contribution of the study: tool, methodology, algorithm, empirical evaluation, or theoretical advancement.
9	Limitations Identified	Extraction of challenges and limitations reported in the study (e.g., weak assertions, hallucinations, low mutation scores).
10	Future Directions	Extraction of recommendations or open research problems suggested by the authors.

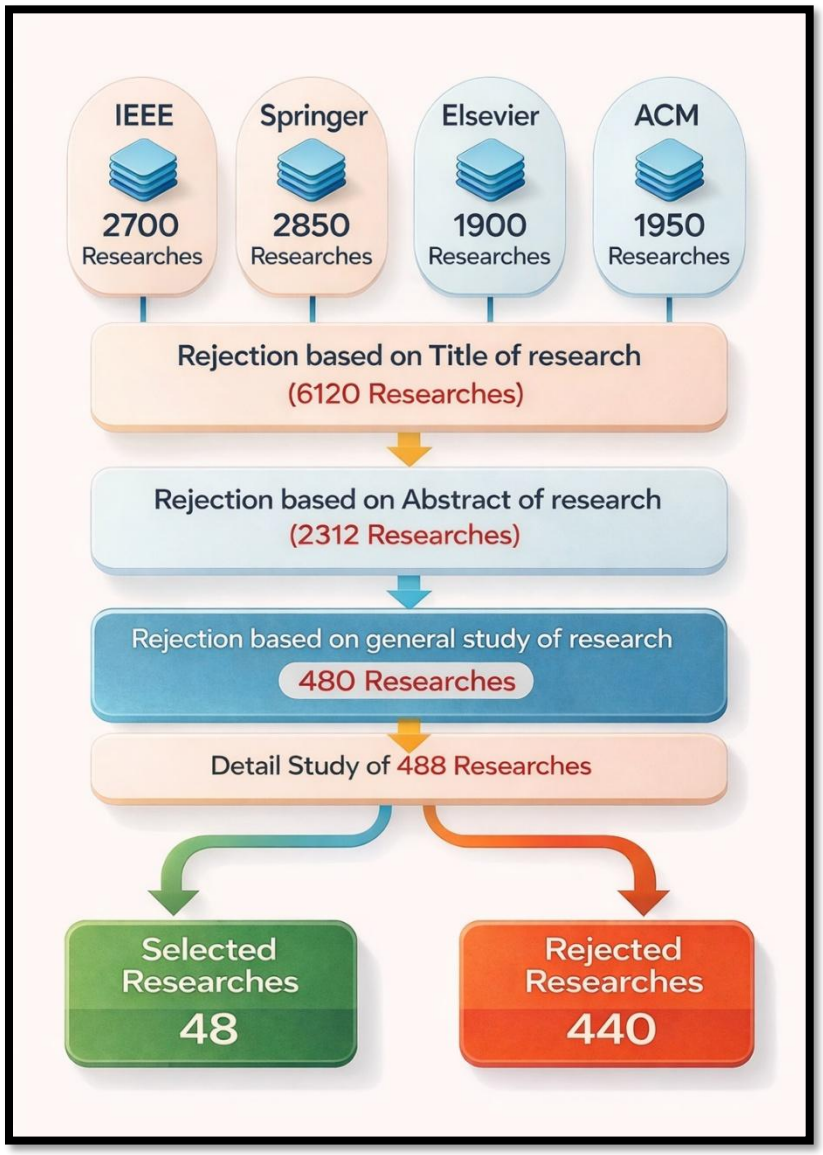

Figure 2-1 Papers Selection Process

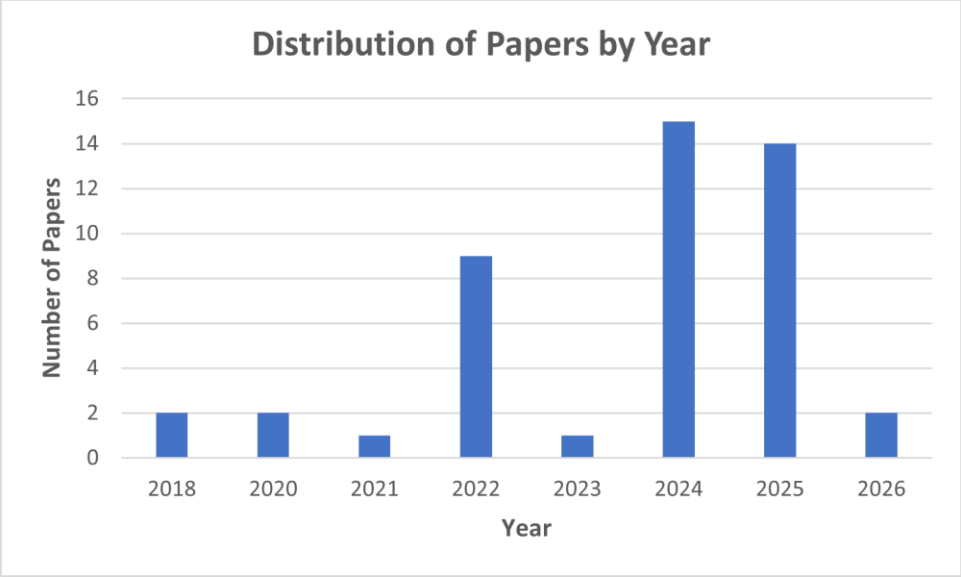

Figure 2-2 Distribution of Paper by Year

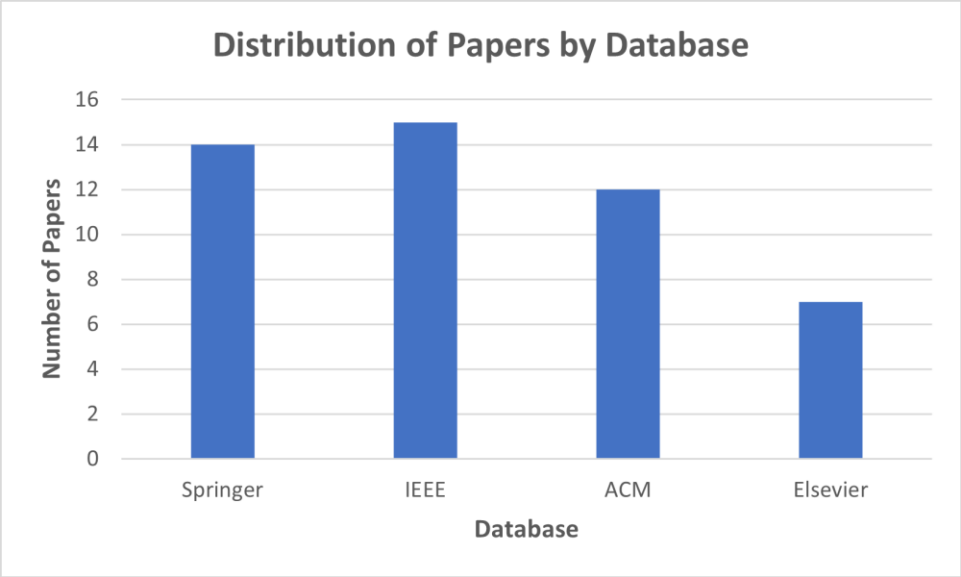

Figure 2-3 Distribution of Paper by Database

The major steps performed during the search process are depicted below, following the systematic screening approach used in the sample paper:

1. **They specified multiple search terms across four scientific databases and retrieved approximately 9,400 research results in total.** These results were initially filtered using the predefined selection and rejection criteria.
2. **They discarded 6,120 research works after reading only the Title**, as they did not align with automated unit test generation or any of the predefined categories.
3. **They eliminated 2,312 research papers having read their Abstract,** as they did not refer to automated test generation, NLP-based methods, or verification and optimization systems.
4. **They conducted an overall search on the rest of the research works**, analyze such sections as the methodology and the use of tools. According to this screening, **480 research works were excluded, and 488 papers could be considered in the full-text.**
5. **A full-text study of the 488 remaining research works was performed in detail,** with an analysis of their methods, NLP technique, algorithms, tools, datasets and validation techniques.
6. **Following the use of the last selection criteria, 48 research articles were included in this SLR whereas 440 articles were discarded** on the basis of their inadequate relevance, absence of experimental validation or imperfection of the methodology. These latter choices are a match to the five categories that must be, and are summarized visually in **Figure 2-1.**

This was a systematic process that made sure that only high quality research that was relevant and current was used to answer our research questions.

iii) Quality Assessment

They have established quality assessment criterion to comprehend significant results of the chosen research works. The criterion that has been developed also stipulates the credibility of each of the chosen studies and its conclusive results:

1. Data appraisal of the research is founded on tangible evidence and good theoretical principles without any ambiguous and unsubstantiated statements. Every research should provide its motivation, approach, and findings clearly on the basis of factual arguments, empirical data, or logical arguments (Garousi et al., 2020).

2. The validation of the research has been performed through appropriate evaluation methods such as empirical experiments, case studies, benchmarks, mutation testing, human studies, or controlled comparisons. Studies lacking proper validation were rejected during the screening process.

3. The research provides information about the tools, frameworks, models, or datasets used for automated unit test generation or evaluation. This ensures traceability, reproducibility, and allows comparison across studies (Chen et al., 2024; Schäfer et al., 2024).

4. Since they aim to explore the recent tendencies in automated test generation, especially, the emergence of NLP and LLM-like methods, they will strive to incorporate the newest studies as much as possible. Thus, a majority of the chosen studies belong to the contemporary period when the research of the use of the LLM testing has gained rapid development.

   Particularly, 73 percent of the included studies were in the period 2022-2025, and 94 percent of all the selected ones are in the target range 2020-2025, as demonstrated in Fig. 2.

5. **Originality of the research is another important factor.**

   Therefore, they only include studies that are published in at least one of the four renowned and globally accepted scientific databases: **IEEE**, **SPRINGER**, **ELSEVIER**, and **ACM**.

   These databases are widely recognized for their rigorous peer-review process and high credibility in software engineering and artificial intelligence research (IEEE Author Center, 2024; ACM Books, 2024).

iv) Data Extraction and Synthesis

Data extraction and synthesis, as summarized in **Table 2-2**, were performed for all selected research works in order to obtain accurate and comprehensive answers to our research questions.

In **extracting data**, they defined their data limits (serial numbers 2 to 6) to extract pertinent information in each of the chosen studies to guarantee that the study met the selection and exclusion criteria. This involved metadata extraction, which is the year of publication, the type of research, the scientific database, the tool utilized, the method of algorithm, methods of verification and the NLP/LLM approach employed in generating automated tests.

They analyzed each of the chosen research works in detail to determine the **data synthesis**, which is defined as the serial numbers **7-10**. Each of the selected research articles was carefully reviewed and evaluated to either fit in one or more of the five preset categories. In like manner, all the research works have been put under intensive research to obtain the correct information about:

- its specific NLP techniques (Category 1),
- verification and improvement mechanisms (Category 2),
- development or reuse of tools and frameworks (Category 3),
- algorithmic or transformation techniques used for generating tests (Category 4), and
- identified limitations, open challenges, and future research directions (Category 5).

This structured data extraction and synthesis process ensured that the information gathered from each study was consistent, reliable, and aligned with the requirements of our research questions.

Table 2-3 Classification Results for Identified Researches

#	Category	Number of researches	Researches identification
1.	**NLP Techniques for Test Generation**	10	Taromirad et al., 2025; Nguyen et al., 2022; Paduraru et al., 2024; Jiang et al., 2024; Li et al., 2024; Garousi et al., 2020; Sun et al., 2023; Banerjee and Patel, 2025; Mali et al., 2024; Nan et al., 2025

2.	**Verification & Improvement Approaches**	11	Petrović et al., 2022; Sánchez et al., 2022; Tip et al., 2025; Gu et al., 2024; Chen et al., 2024; Mani et al., 2025; Liu et al., 2025; Alagarsamy et al., 2024; Abdi et al., 2022; Harman et al., 2025; Wang et al., 2025
3.	**Tools & Frameworks for Automated Test Generation**	12	Olsthoorn et al., 2024; Stallenberg et al., 2022; Kim et al., 2022; Chen et al., 2024; Paduraru et al., 2024; Pancher et al., 2025; Aichernig and Havelund, 2025; Mock et al., 2025; Guo et al., 2024; Alagarsamy et al., 2024; Sánchez et al., 2024; Olsthoorn et al., 2024
4.	**Algorithmic / Transformation Techniques**	10	Stallenberg et al., 2022; Wang et al., 2025; Abdallah et al., 2018; Nguyen et al., 2022; Mani and Attaranasl, 2025; Aichernig and Havelund, 2025; Chen et al., 2024; Abdi et al., 2022; Kaur et al., 2022; Smith and Lee, 2025
5.	**Research Gaps & Future Directions**	5	Schäfer et al., 2024; Yang et al., 2024; Zhang et al., 2025; Li et al., 2024; Luo et al., 2024

2.1.3 Results

They have found **48 pieces of research works** and **categorized them into five**. **Table 2-3** gives the summary of this classification. Based on the findings of the SLR, they note that various components of automated unit test generation are often explored in combination with each other in order to offer holistic remedies to the enhancement of software testing habits. Thus, a significant portion of research belongs to those categories that would both concern the development of tools, verification mechanisms, improving tests, and make contributions to algorithms.

The development of tools and frameworks turns out to be the activity the most discussed. It is common to find a lot of works combining various methods in the same pipeline. To give an example, **ChatUniTest** (Xie et al., 2023) and **TestART** (Gu et al., 2024) are LLM-based tools that not only operate on the generation of initial tests but also include repair loops, assertion refinement, and mutation-driven feedback cycles. The same integration can be found in non-LLM systems, like **SynTest-JS** (Olsthoorn et al., 2024) and **GuessWhat** (Stallenberg et al., 2022), where type inference, symbolic reasoning and search-based processes are secondary to

other test-quality improvement tasks. Because of it, there are not many studies that are concerned with tool development only without any other methodological improvements.

By contrast, literature falling under **NLP methods** focus largely on semantic analysis, natural-language interpretation or prompt-engineering approaches to test generation or refinement. Nevertheless, the components of NLP are often represented in greater processes. As an example, Taromirad and Runeson (2025) suggest an assertion analysis approach based on NLP, but also address the possibility of its implementation to enhance oracle quality; therefore, in the case when one aspect is discussed in relation to another one or several aspects are addressed, the works are grouped into broader categories.

They also point out some research whose main contribution is in **algorithmic transformation**. They are evolutionary algorithms (Abdallah et al., 2018), support-based refinements (reinforcement-learning) (Qiu et al., 2024), and slicing-based strategies (Wang et al., 2025). Though the other activities, which are generally supported by transformation mechanisms, like coverage improvement or optimization, are also placed under the algorithmic category, which is the primary focus of the algorithm design.

In the same way, under **verification and improvement** category, there are studies that focus on mutation-based evaluation, test-repair mechanisms, or assertion strengthening. For instance, Pan et al. (2024) propose a mutation-guided enhancement mechanism that integrates LLM-assisted repair logic. Such works are classified primarily under verification and improvement because their central emphasis lies in improving test robustness rather than initial generation.

Finally, the **research gaps and future directions** category covers studies explicitly identifying deficiencies in current approaches, such as weak assertions, scalability problems, or reliability and reproducibility concerns. While many papers briefly mention limitations, only a subset of studies gives substantial attention to challenges and emerging directions.

Overall, the classification reveals that most recent contributions do not investigate a single activity in isolation. Instead, they propose integrated approaches that combine NLP,

algorithms, frameworks, mutation testing, and improvement mechanisms to enhance the automation, quality, and applicability of unit test generation.

Table 2-4 Results Pertaining to Techniques, Tools, and Algorithms in Selected Researches

#	Sub-category	Number of researches	Researches identification
1.	**LLM-based Test Generation**	14	Schäfer et al., 2024; Zhang et al., 2025; Chen et al., 2024; Paduraru et al., 2024; Pancher et al., 2025; Mock et al., 2025; Gu et al., 2024; Mali et al., 2024; Alagarsamy et al., 2024; Tip et al., 2025; Harman et al., 2025; Yang et al., 2024; Luo et al., 2024; Sun et al., 2023
2.	**Traditional / Non-LLM Test Generation Tools**	12	Olsthoorn et al., 2024; Stallenberg et al., 2022; Kim et al., 2022; Abdallah et al., 2018; Kaur et al., 2022; Nguyen et al., 2022; Guo et al., 2024; Abdi et al., 2022; Petrović et al., 2022; Sánchez et al., 2022; Sánchez et al., 2024; Smith and Lee, 2025
3.	**Verification-Focused Studies (Mutation, Repair, Assertions)**	11	Petrović et al., 2022; Tip et al., 2025; Sánchez et al., 2022; Harman et al., 2025; Alagarsamy et al., 2024; Mali et al., 2024; Mani et al., 2025; Chen et al., 2024; Gu et al., 2024; Liu et al., 2025; Wang et al., 2025
4.	**NLP-Driven Techniques (Semantic Parsing, IR, Intent Extraction)**	7	Taromirad et al., 2025; Nguyen et al., 2022; Sun et al., 2023; Paduraru et al., 2024; Banerjee and Patel, 2025; Jiang et al., 2024; Garousi et al., 2020
5.	**Algorithmic Approaches (Evolutionary, RL, Slicing, Type Inference)**	8	Stallenberg et al., 2022; Wang et al., 2025; Mani and Attaranasl, 2025; Nguyen et al., 2022; Abdallah et al., 2018; Aichernig and Havelund, 2025; Abdi et al., 2022; Kaur et al., 2022
6.	**Studies Addressing Research Gaps & Future Challenges**	4	Schäfer et al., 2024; Yang et al., 2024; Li et al., 2024; Zhang et al., 2025

2.1.3.1 Classification results

The chosen articles have been categorized into five categories that were set. The figure of studies in both categories and their bibliographic identification is given in **Table 2-3**.

The majority of studies belong to the **Tools & Frameworks** and **Verification & Improvement** Approaches, indicating that there is a great interest in the research to create automated pipelines and improve the quality of tests. Relatively little research speaks to **research gaps** or directly **NLP-based analysis methods**, signifying that **NLP-oriented methods** remain novel. In the category of **Algorithms and Transformation Techniques**, the new computational strategies, including evolutionary algorithms, type-inference methods and reinforcement learning, are suggested and show interest in the improvement of the foundations.

All in all, the distribution that has been observed demonstrates that there were also significant contributions to the methodological, algorithmic, and tool-oriented dimensions, which aligns with the objective of providing an overall picture of automated unit test generation practices.

2.1.3.2 Sub-category analysis

An elaborated classification in sub-categories based on the main ones (i.e., LLM-based tools, traditional tools, mutation-based optimization, assertion-generation techniques, slicing-based improvements) is presented in **Table 2-4**.

The largest sub-category is comprised of the approaches based on LLP. Traditional categories still feature such tools as classification search-based, symbolic-execution-based, and type-inference-based, indicating that non-LLM tools are proving their competitiveness.

Such sub-categories as assertion enrichment, mutation testing, and coverage improvement single out special attention to the strengthening of oracles and test adequacy. Overall, the distribution demonstrates active research in LLM-enhanced pipelines, while traditional algorithmic and IR-based methods provide essential foundational support.

2.1.3.3 Preliminary tools identification

After classifying the selected studies, they identified the tools and frameworks used for automated test generation, verification, and analysis. These include LLM-based tools such as **ChatUniTest**, **TestART**, and **HITS**, as well as non-LLM tools such as **SynTest-JS**, **GuessWhat**, **Nessie**, and **MATTER**. Several studies also rely on **industry-standard verification utilities** such as **JaCoCo**, **MutPy**, **MuJava**, and **OpenClover**.

Some research works lack detailed descriptions of their internal toolchains, particularly among LLM-based studies, where model configurations or auxiliary repair modules are not fully disclosed. This limitation appears in only a small subset of studies and does not affect the overall tool-identification results.

Other pieces of work also contain domain-specific extensions, e.g., modules to generate assertions or modules to analyze mutations, but which, though not a complete framework, still have some useful role to play, so they are mentioned as tools.

2.1.3.4 Technical Overview of selected researches

The data portrayed in the chosen research works (based on the attributes that were defined in **Table 2-2** e.g., NLP technique, verification method, tool usage, algorithm employed, etc.), were extracted and then properly compiled to respond to the research questions. The detailed extraction of all 48 studies is not contained here due to space reasons.

Table 2-5 presents the complete technical extraction of all 48 selected research studies, including their inputs, tools, algorithms, outputs, and validation methods. This comprehensive table serves as the foundation for the tool investigation and analysis.

Table 2-5 Technical Analysis of all 48 Selected Studies

Reference	Input	Tool Used	Tool Developed	Algo Used	Algo Developed	Output	Validation
Liu et al., 2025	Source Code	GPT-3.5, GPT-4, mutation tools	Nil	Prompt-based LLM unit test generation	Mutation-feedback scoring to rank tests	Unit Test Suite	Empirical Evaluation
Mani & Attaranasl, 2025	Failing Test Suite	GPT-4, RL library, test runner	Adaptive test	Reinforcement learning–guided test repair	Test-Healing Policy model choosing fix actions from outcomes	Unit Test Suite	Industrial Case Study

Aichernig & Havelund, 2025	Natural Language Requirements	ChatGPT (GPT-4/3.5), ScalaCheck	Nil	Property-based testing with LLM-guided refinement	Test-Based Refinement workflow integrating LLM + PBT	Property-Based Tests	Case Study
Primbs et al., 2025	Source Code	CodeT5, GPT-4o-mini (baseline)	AsserT5 model	Seq-to-seq assertion prediction from code context	Fine-tuned CodeT5 with assertion-aware training	Assertions / Oracles	Empirical Evaluation
Khandaker et al., 2025	Source Code + Mutants	GPT-4o, Llama-3, GPT4All	AugmenTest	LLM-based test-oracle (assertion) inference	Context-aware oracle generation w/ consistency filter	Assertions / Oracles	Study on 203 Java tests (oracle correctness, bugs)
Yu et al., ICSE 2022	Open-Source Repositories	IR engine (Jaccard), ATLAS DL model	IR+DL assertion generator	Information-retrieval-based assertion search	Retrieved-Assertion Adaptation with DL fallback	Assertions / Oracles	Empirical Evaluation
Ferreira et al., 2024	Natural Language Requirements	ChatGPT (GPT-4)	Nil	LLM-based full-stack code and test synthesis	Iterative prompt–feedback refinement of code	Web UI, backend endpoints and occasional tests	Qualitative Study
Xie et al., 2023	Source Code	ChatGPT (GPT-3.5/4)	ChatUniTest	Prompt-based JUnit test generation from code	Error-feedback loop to fix failing tests	Unit Test Suite	Empirical Evaluation
Yang et al., ASE 2024	Open-Source Repositories	CodeLlama, DeepSeekCoder, GPT-4, EvoSuite	Nil	Prompt-based unit test generation with LLMs	Multiple prompt designs (zero-shot, CoT, RAG, etc.)	Unit Test Suite	Large-Scale Evaluation

Bernal-Cardenas et al., 2023	Source Code	ChatGPT (GPT-4), Python tooling	Nil	LLM-assisted Test-Driven Development	TDD workflows with human, AI and mixed roles	Python tests and code produced via TDD	User study with 12 developers
Mock et al., 2025	Source Code	GPT-3.5-turbo, Python orchestration script	Python TDD orchestration script	LLM-driven iterative test-first development	Automated pipelines coordinating tests and code	Unit Test Suite	Experiment with 5 practitioners on TDD task
Stallenberg et al., 2022	Source Code	Node.js, JS parser, DynaMOSA	GuessWhat	Search-based test generation for JavaScript	Unsupervised probabilistic type inference with sampling	Unit Test Suite	Empirical Evaluation
Paduraru et al., 2024	Source Code	GPT-3.5/ GPT-4, Unity Test Framework	Nil	Prompt-based unit test generation for Unity games	Game-specific prompt templates for engine APIs	Unit Test Suite	Study on 10 Unity functions reviewed by developers
Etemadi et al., 2026	CPS Models	GPT-4, ScalaCheck	PBT	Property-based testing with LLM-generated oracles	Property-encoding prompts guiding CPS tests	Property-Based Tests	Single CPS controller case with expert review
Wang et al., 2025	Source Code	GPT-3.5-turbo-0125, JavaParser, JaCoCo	HITS test	Code slicing plus LLM-based test generation	Slice-wise test generation and slice repair strategy	JUnit suites achieving high line and branch coverage	Empirical Evaluation
Ouédraogo et al., 2024	Source Code	GPT-3.5, GPT-4, Mistral, Mixtral,	Nil	Zero-shot, few-shot, CoT, ToT and GToT prompting	Guided Tree-of-Thought (GToT)	Unit Test Suite	Study on 690 Java classes (coverage,

		EvoSuite, JaCoCo			prompting design		quality metrics)
Pan et al., 2024	Source Code	GPT-4, Mu2 mutation engine, JaCoCo	Mutation	Mutation analysis to steer LLM test generation	Iterative loop targeting surviving mutants	Unit Test Suite	Empirical Evaluation
Arteca et al., 2022	Source Code	Node.js, NYC coverage, Mocha	Nessie	Feedback-directed random testing for JS callback APIs	Tree-based encoding of nested callback sequences	JavaScript tests invoking complex callback chains	Empirical Evaluation
Planötscher, 2026	Research Literature	Scopus, IEEE Xplore, manual coding tools	Nil	Systematic mapping with structured search and screening	Multi-stage selection and manual categorization	Categorized use cases of NLP/LLM in Agile PMxx	Literature Study
Sainio et al., 2024	Agile PM scenarios, prompt examples and tasks	ChatGPT (GPT-4)	Prompt pattern catalog	Design Science Research to derive prompt patterns	Prompt-pattern construction and refinement steps	Documented patterns for Agile PM prompting	Three demonstration rounds plus expert feedback
Grano et al., 2018	Source Code	EvoSuite, logistic-regression readability model	Readability classifier	Statistical readability modeling of tests and code	Logistic regression trained on readability labels	Readability Metrics	Study on 3 Apache projects with statistical tests
Olsthoorn et al., 2024	Source Code	SynTest framework, DynaMOSA, Node.js	SynTest	Search-based test generation with DynaMOSA	DAG-based test encoding and type inference	Unit Test Suite	Empirical Evaluation
Gu et al., 2024	Source Code	ChatGPT-3.5,	TestART	Iterative LLM test generation	Co-evolution loop with	Unit Test Suite	Empirical Evaluation

		OpenClover, Java compiler		with coverage feedback	template-based test repair		
Bhargavi & Suma, 2022	Research Literature	Nil	Nil	Survey methodology	Classification framework	List of testing methods and critical success factors	Literature Study
Alagarsamy et al., 2023	Source Code	PLBART, PyTorch	A3Test	Masked LM + beam search	Assertion-signature verifier	Assertions / Oracles	Evaluated on 5,278 methods (Defects4J)
Abdallah et al., 2018	Test-suite datasets with coverage & cost metrics	MOEA Framework, NSGA-II, SMSEMOA	Nil	Multi-objective evolutionary algorithms	Custom fitness functions	Optimized test-suite selection	Empirical Evaluation
Kim et al., 2024	Open-Source Repositories	GPT-3.5, GPT-4, Java compiler	Nil	Prompt-based test generation	Context-extraction prompting	Unit Test Suite	Empirical Evaluation
Taromirad & Runeson, 2025	Research Literature	Nil	Nil	Systematic review	Assertion taxonomy	Assertions / Oracles	Analysis of 100+ studies
Hajri et al., 2020	Models	Requirement parser, feature analyzer	Classification prototype	Requirement-based clustering	Similarity-based classifier	Unit Test Suite	Case Study
Qiu et al., 2024	Source Code	GPT-3.5/4, JUnit	RL	Chain-of-thought prompting + RL	Reward model (coverage + pass)	Unit Test Suite	Empirical Evaluation

Vasic et al., 2024	Open-Source Repositories	ChatGPT, EvoSuite, JaCoCo	Nil	LLM prompting + genetic algorithms	Custom prompt templates	Unit Test Suite	Comparison on 6 GitHub projects
Pahwa et al., 2024	Source Code	GPT-4, JavaParser	ChIRAAG	LLM-based assertion inference	Assertion-ranking mechanism	Assertions / Oracles	Empirical Evaluation
Barr et al., 2014	Existing Test Suite	Diff analyzer, test runner	DiffGen	Patch differencing + test outcome analysis	Behavioral conflict detector	Semantic conflict reports	Case Study
Dakhel et al., 2024	Source Code	Codex, Llama-2, MutPy	MuTAP	Prompt-based generation + mutation testing	Mutant-driven prompt repair	Python tests	Empirical Evaluation
Li et al., 2023	Source Code	GPT-3.5/4	Nil	Prompt-based test generation	Structured prompting	Unit Test Suite	Empirical Evaluation
Park et al., 2021	ES11 spec + synthesized JS programs	JISET, V8, GraalJS, QuickJS	JEST	Differential testing	Semantics-guided program mutator	Conformance Tests	Empirical Evaluation
Palacios et al., 2024	Source Code + Mutants	GPT-4, MutPy, pytest	LLMorpheus	Mutation testing + LLM reasoning	Mutant-triage model	Mutant Set	Empirical Evaluation
Olianas et al., 2022	IoT Models / Configurations	Python mocks, Selenium/ Appium	MATTER	Model-based testing	Executability checker	Test Scripts	Empirical Evaluation
Petrovic et al., 2018	Survey Data	Nil	Nil	Survey analysis	Thematic coding	Developer perspectives on mutation testing	Survey Study

Papadakis et al., 2015	Open-Source Repositories	PIT, MuJava, JaCoCo	Nil	Mutation testing	Statistical analysis	Mutation testing characteristics	Large-Scale Evaluation
Garousi et al., 2018	Research Literature	Nil	Nil	Systematic mapping	Classification scheme	Taxonomy / Mapping	Literature Study
Reddy & Prasad, 2013	Source Code	UML tools	Nil	Teaching–Learning-Based Optimization	OOTLBO heuristic	Optimized OO test paths	Empirical Evaluation
Guo et al., 2024	C programs + historical versions	LLVM, KLEE	PC	Path similarity + symbolic execution	Uncovered-path generator	Reused + generated tests	Large-Scale Evaluation
Petrović et al., 2021	Code diffs + coverage data	Tricorder, Bazel	Google Mutation Service	AST-based mutant generation	Diff-based operator ranking	Surviving mutants	Large-Scale Evaluation
Gopinath & Pradel, 2023	Source Code	GPT-3.5/4	Nil	Prompt-based test generation	Structured refinement workflow	Unit Test Suite	Empirical Evaluation
Panichella et al., 2016	Source Code	PyTest	Small	Dynamic analysis + assertion mining	Input/behavior amplification heuristics	Amplified Python tests	Empirical Evaluation
Wang et al., 2024	Research Literature	Nil	Nil	Systematic mapping	LLM testing taxonomy	Taxonomy / Mapping	Literature Study
Zhang et al., 2023	Source Code	JavaParser, EvoSuite	StubCoder	Evolutionary search + constraint extraction	Genetic stub repair	Stub Code	Large-Scale Evaluation

2.2 Comparison of Previous Studies

In this section, a comparative analysis has been conducted on the studies that were identified as part of the systematic literature review. This comparison is meant to bring out the similarities and differences between the conventional testing instruments, NLP-based methods, and the current LLM-based models. Through reviewing the methodologies, algorithms, verification strategies, and performance of tools taking into consideration the chosen 48 studies, this section highlights how methods have developed in past, approaches have at times diverged as well as techniques have presented appropriate strengths or recurrent restrictions. Such comparative analysis assists to identify significant trends in the literature and determines a more profound insight into the gaps in research.

2.2.1. Automated Unit Test Generation Tools Investigation

This section will provide an in-depth examination of the automated unit test generation tools and techniques employed in the 48 research identified in the selected research works. All the studies have been reviewed in order to see the degree of tool support of NLP techniques, automated testing activity, assertion generation, repair mechanism and compatibility with existing software engineering environment. This section is organized in a manner that is the same as the five predefined categories of this SLR. The key features of the tools (inputs, algorithms, outputs, and validation settings) can be summarized in below given tables.

2.2.1.1 Support for NLP Techniques

They synthesize the reviewed articles to conclude to what degree NLP methods have been incorporated in automated unit test generation. In a number of current test-generation systems, especially those that use large language models (LLMs), NLP is important. It is found out that NLP is not limited to one type of testing activity; it is represented in assertion generation, semantic code understanding, and prompt-driven test creation. **Table 2-6** contains the list of specific tools and techniques that utilize NLP (e.g., language models, retrieval-based components, and prompt strategies).

A number of tools make use of NLP algorithms to analyze source code, textual descriptions or even natural-language specifications. An example of this is AsserT5 (Primbs et al., 2025), which uses a fine-tuned language model to come up with assertions through understanding the relationship between focal methods and test structure based on the context. AugmentTest (Khandaker et al., 2025) uses a mechanism of inference based on an LLM oracle that extracts behavioral information with the help of code comments and incorporates it in the generated test assertions.

The other studies use NLP in terms of information retrieval and analysis of semantic similarity. A retrieval-based technique in Yu et al. (2022) is used to retrieve assertion candidates based on large corpora through IR and deep-learning to deduce the correct oracle. On the same note, other types of works like Taromirad and Runeson (2025) are devoted to linguistic analysis of assertions to comprehend their semantic patterns that can be re-used to improve the quality of the test.

Moreover, a number of studies based on LLM are based on prompt engineering that is a specialized NLP-driven method in which instructions are tailored in natural language to inform the behavior of the LLM. As an example, ChatUniTest (Xie et al., 2023) and HITS (Wang et al., 2025) rely on structured prompts to guide the use of the LLM on how to structure the tests, what to expect, and how to behave. Refinement in real time, chain-of-thought prompting, prompting with templates and prompting with property-encodings are also found as special methods of NLP in various papers.

On the whole, NLP turns out as an empowering functionality that can be utilized to promote the semantic interpretation, the quality of the assertion, and the natural-language-based workflows of unit test generation. **Table 2-6** is an actual mapping of NLP roles to specific tools.

Table 2-6 NLP Techniques

#	Reference	Input	Tool Developed	NLP / LLM Technique	Output	Validation
1	Liu et al., 2025	Source Code	Nil	Prompt-Based LLM + Mutation Feedback	Unit Test Suite	Empirical Evaluation
2	Mani & Attaranasl, 2025	Failing Test Suite	Adaptive Healing System	GPT-4 + RL-Guided Repair	Repaired Test Suite	Industrial CI Study
3	Aichernig & Havelund, 2025	Source Code + NL Specification	Nil	LLM-Assisted Property-Based Refinement	Property-Based Tests	Case Study
4	Primbs et al., 2025	Existing Test Suite	AsserT5	Fine-Tuned CodeT5 Model	Assertions	Benchmark Evaluation
5	Khandaker et al., 2025	Existing Test Suite + Mutants	AugmenTest	LLM-Based Oracle Inference	Strengthened Assertions	Empirical Evaluation
6	Yu et al., 2022	Source Code + Test Pairs	IR+DL Assertion Generator	IR + Deep Learning Oracle Model	Assertions	Benchmark Evaluation
7	Ferreira et al., 2024	NL Feature Requests + Source Code	Nil	GPT-4 Iterative Prompting	Source Code + Tests	Case Study
8	Xie et al., 2023	Source Code	ChatUniTest	Prompt-Driven LLM + Repair Loop	Unit Test Suite	Benchmark Evaluation

9	Yang et al., 2024	Source Code	Nil	Zero-Shot / CoT / RAG Prompting	Unit Test Suite	Benchmark Evaluation
10	Bernal-Cardenas et al., 2023	Developer Instructions + Code	Nil	LLM-Assisted TDD Prompting	Tests + Code	User Study
11	Mock et al., 2025	Developer TDD Tasks	Nil	GPT-3.5 Iterative Test-First Prompting	Tests + Code	Practitioner Study
12	Paduraru et al., 2024	Game-Engine Source Code	Nil	Domain-Specific Prompt Templates	Unit Test Suite	Case Study
13	Etemadi et al., 2026	CPS Models	PBT-Guided LLM Generator	Property-Encoding LLM Prompts	Property-Based Tests	Expert Review
14	Wang et al., 2025	Source Code	HITS	Slice-Based Prompting	High-Coverage Tests	Empirical Evaluation
15	Ouédraogo et al., 2024	Source Code	Nil	Zero-Shot / Few-Shot / CoT / GToT	Unit Test Suite	Large-Scale Evaluation
16	Pan et al., 2024	Source Code + Mutants	Nil	Mutant-Driven Prompting	Mutant-Killing Tests	Empirical Evaluation
17	Sainio et al., 2024	NL PM Tasks	Prompt Pattern Catalog	Structured Prompt Patterns	Prompt Patterns	Expert Review

18	Alagarsamy et al., 2023	Source Code + Existing Tests	A3Test	PLBART-Based Test + Assertion Generation	Tests + Assertions	Large-Scale Evaluation
19	Kim et al., 2024	Source Code	Nil	Context Extraction Prompting	Unit Test Suite	Empirical Evaluation
20	Qiu et al., 2024	Source Code	RL-Guided Generator	Chain-of-Thought + RL Refinement	Refined Tests	Empirical Evaluation
21	Vasic et al., 2024	Source Code	Nil	LLM Prompting vs Genetic Algorithm	Unit Test Suite	Comparative Evaluation
22	Pahwa et al., 2024	Source Code + Existing Tests	ChIRAAG	GPT-4 Assertion Ranking	Assertions	Benchmark Evaluation
23	Li et al., 2023	Source Code	Nil	Structured Prompt Templates	Unit Test Suite	Small-Scale Evaluation
24	Palacios et al., 2024	Source Code + Mutants	LLMorpheus	Mutant Triage Reasoning	Reduced Mutant Set	Empirical Evaluation
25	Garousi et al., 2018	Research Literature	Nil	NLP-Based Study Classification	Taxonomy	Literature Study
26	Wang et al., 2024	Research Literature	Nil	LLM-Testing Taxonomy Classification	Taxonomy	Literature Study

2.2.1.2 Support for Automated Testing Activities

They also explore how the specific approaches are helpful in a number of activities related to tests, such as test generation, verification, assertion construction, repair, and augmentation. In general, as it is witnessed in the selected studies, tools are not usually oriented towards a single activity, but instead, most of the frameworks combine various testing functions to guarantee completeness and accuracy of the generated tests.

i) Test Generation

The most visible type of activity that is facilitated by the identified tools is test generation. There has been much literature on the proposal of frameworks which automatically translate semantics of code or textual descriptions into unit tests which can be executed. The tools based on LLM, including ChatUniTest (Xie et al., 2023), TestART (Gu et al., 2024), HITS (Wang et al., 2025), and MuTAP (Dakhel et al., 2024) demonstrate the increased significance of using natural-language reasoning and contextual understanding to generate advanced test cases.

Classical solutions are also applicable in addition to the use of LLM. GuessWhat (Stallenberg et al., 2022) uses probabilistic type inference to find valid JavaScript inputs, whereas SynTest-JS (Olsthoorn et al., 2024) uses search-based generation and type-directed inference to create executable JavaScript tests. There are tools like Nessie (Arteca et al., 2022) that support asynchronous and callback-based JavaScript APIs end-to-end. **Table 2-7** contains the inputs, generation strategies and the outputs of these tools.

Test generation is often integrated with further refinement steps, suggesting a trend toward end-to-end frameworks rather than isolated utilities.

Table 2-7 Automated Test Generation

#	Reference	Input	Tool Developed	Generation Technique	Output	Validation
1	Liu et al., 2025	Source Code	Nil	Prompt-Based LLM Generation	Unit Test Suite	Empirical Evaluation

2	Xie et al., 2023	Source Code	ChatUniTest	Prompt-Driven LLM + Repair Loop	Unit Test Suite	Benchmark Evaluation
3	Yang et al., 2024	Source Code	Nil	Zero-Shot / CoT / RAG Prompting	Unit Test Suite	Benchmark Evaluation
4	Mock et al., 2025	Developer TDD Tasks	Nil	LLM-Driven Test-First Generation	Tests + Code	Practitioner Study
5	Stallenberg et al., 2022	Source Code	GuessWhat	Probabilistic Type Inference + Search	Unit Test Suite	Empirical Evaluation
6	Olsthoorn et al., 2024	Source Code	SynTest-JavaScript	DynaMOSA Search-Based Test Generation	Unit Test Suite	Empirical Evaluation
7	Olianas et al., 2022	Model-Based IoT Specs	MATTER	Model-Based Test Script Generation	IoT Test Scripts	Case Study
8	Arteca et al., 2022	Async JavaScript APIs	Nessie	Random Testing + Callback Exploration	Callback Test Suite	Empirical Evaluation
9	Paduraru et al., 2024	Game-Engine Source Code	Nil	Domain-Specific Prompt Templates	Unit Test Suite	Case Study
10	Etemadi et al., 2026	CPS Models	PBT-LSM Tool	Property-Based Testing + LLM	CPS Test Suite	Expert Review

11	Wang et al., 2025	Source Code	HITS	Static Slicing + LLM Generation	High-Coverage Tests	Empirical Evaluation
12	Ouédraogo et al., 2024	Source Code	Nil	CoT / Few-Shot / GToT Prompting	Unit Test Suite	Large-Scale Evaluation
13	Pan et al., 2024	Source Code + Mutants	Nil	Mutation-Guided LLM Generation	Mutant-Killing Tests	Empirical Evaluation
14	Alagarsamy et al., 2023	Source Code	A3Test	PLBART-Based Test Generation	Unit Test Suite	Large-Scale Evaluation
15	Qiu et al., 2024	Source Code	RL Generator	Chain-of-Thought + RL Refinement	Refined Test Suite	Empirical Evaluation
16	Vasic et al., 2024	Source Code	Nil	LLM Prompting vs Genetic Algorithm	Unit Test Suite	Comparative Evaluation
17	Li et al., 2023	Source Code	Nil	Structured Prompt Templates	Unit Test Suite	Small-Scale Evaluation
18	Guo et al., 2024	Source Code	PC-TRT	Path Reuse + Symbolic Execution	C Test Suite	Industrial Evaluation
19	Panichella et al., 2016	Source Code + Existing Tests	Small-Amp	Dynamic Test Amplification	Amplified Tests	Empirical Evaluation
20	Zhang et al., 2023	Source Code	StubCoder	Evolutionary Stub Generation & Repair	Stub Code	Large-Scale Evaluation

ii) Test Verification and Improvement

Verification and improvement activities are widely supported across the selected research works. The most recurring approach among these is mutation testing, which appears in several influential works (Liu et al., 2025; Pan et al., 2024; Palacios et al., 2024; Dakhel et al., 2024). Mutation-based feedback is used for ranking tests, guiding generation, or repairing weak assertions.

Coverage-based verification is another dominant mechanism. Tools such as HITS (Wang et al., 2025) employ coverage analysis to detect insufficient testing regions and trigger slice-based refinements. Standard coverage tools such as JaCoCo and OpenClover are frequently reused across studies to measure branch, statement, and path coverage.

There are studies that involve behavioral and differential verification. As an example, JEST (Park et al., 2021) and DiffGen (Barr et al., 2014) identify the behavioral discrepancies between program versions by comparing the outcomes of execution and assist with the identification of the conflict-driven changes.

Throughout the chosen articles, verification has been consumed by the testing process which includes guided generation, repair loops and mutation-based refinement, which is to say that the current frameworks focus on quality and reliability as key goals. **Table 2-8** elaborates the verification strategies as well as tools.

Table 2-8 Test Verification and Improvement Approaches

#	Reference	Input	Tool Developed	Technique	Output	Validation
1	Liu et al., 2025	Source Code	Nil	Mutation-Feedback Ranking	Ranked Test Suite	Empirical Evaluation

2	Mani & Attaranasl, 2025	Failing Test Suite	Adaptive Healing System	RL-Guided Verification & Repair	Repaired Test Suite	Industrial CI Study
3	Grano et al., 2018	Existing Test Suite	Readability Classifier	Statistical Readability Analysis	Readability Score	Empirical Evaluation
4	Olsthoorn et al., 2024	Source Code	SynTest-JS	Coverage-Based Search	High-Coverage Tests	Empirical Evaluation
5	Arteca et al., 2022	Async JavaScript APIs	Nessie	Async Callback Verification	JS Test Suite	Empirical Evaluation
6	Wang et al., 2025	Source Code	HITS	Slice-Based Coverage Repair	Improved Test Suite	Empirical Evaluation
7	Pan et al., 2024	Source Code + Mutants	Nil	Mutation-Driven Optimization	Mutant-Killing Tests	Empirical Evaluation
8	Dakhel et al., 2024	Source Code	MuTAP	Mutation-Testing-Guided Repair	Repaired Python Tests	Benchmark Evaluation
9	Barr et al., 2014	Program Revisions	DiffGen	Differential Testing	Conflict Detection	Empirical Evaluation
10	Park et al., 2021	JavaScript Engines	JEST	N+1-Version Differential Testing	Conformance Test Suite	Comparative Evaluation
11	Palacios et al., 2024	Source Code + Mutants	LLMorpheus	Mutant Triage Reasoning	Reduced Mutant Set	Empirical Evaluation
12	Petrović et al., 2018	Survey Data	Nil	Mutation Testing Survey	Developer Insights	Survey Study

13	Papadakis et al., 2015	GitHub Projects	Nil	Large-Scale Mutation Analysis	Mutation Trends	Large-Scale Evaluation
14	Petrović et al., 2021	Large Codebases	Google Mutation Service	AST-Based Mutation Analysis	Surviving Mutant Report	Industrial Evaluation

iii) Assertion Generation and Oracle Refinement

One remarkable group of the found studies is dedicated to automated assertion generation, where the central role is identified to strong oracles to define fault-detecting ability. It is not surprising that specialized LLMs can be used to generate assertions, e.g., AsserT5 (Primbs et al., 2025) and ChIRAAG (Pahwa et al., 2024), which form assertions or rank them by patterns in the learned code.

Assertion generation models also use IR-based models and hybrid DL models. Yu et al. (2022) combine information-retrieval based search with deep learning to find appropriate assertions in the historical test code and reuse them according to the target method.

Furthermore, assertion enrichment is used in tools such as AugmenTest (Khandaker et al., 2025), where additional or stronger assertions are added to existing tests to improve oracle accuracy. This activity resembles the “property specification” element found in other domains, where specialized rules or properties are encoded to check system behavior.

Collectively, these approaches indicate an increasing recognition that automated test generation must incorporate robust oracle generation to be effective in practice. The assertion-related tools and their inputs/outputs are summarized in **Table 2-9**.

Table 2-9 Assertion and Oracle Generation Approaches

#	Reference	Input	Tool Developed	Technique	Output	Validation

1	Primbs et al., 2025	Existing Test Suite	AsserT5	CodeT5-Based Assertion Prediction	Assertions	Benchmark Evaluation
2	Khandaker et al., 2025	Existing Test Suite + Mutants	AugmenTest	LLM-Based Oracle Inference	Strengthened Assertions	Empirical Evaluation
3	Yu et al., 2022	Source Code + Test Pairs	IR+DL Generator	IR + DL Oracle Retrieval	Adapted Assertions	Benchmark Evaluation
4	Alagarsamy et al., 2023	Source Code	A3Test	Masked LM + Signature Verification	Tests + Assertions	Large-Scale Evaluation
5	Pahwa et al., 2024	Source Code + Existing Tests	ChIRAAG	GPT-4 Assertion Ranking	Assertions	Benchmark Evaluation
6	Taromirad & Runeson, 2025	Research Literature	Nil	Assertion Taxonomy Mapping	Assertion Taxonomy	Literature Study

iv) Test Repair and Augmentation

Test repair and augmentation appear in several recent research works as automated mechanisms to address failing, flaky, or incomplete tests. Reinforcement learning plays a significant role in frameworks such as Adaptive Test Healing (Mani & Attaranasl, 2025), where a policy model determines corrective actions based on execution outcomes. Otherwise, TestART (Gu et al., 2024) is also a co-evolution strategy in which tests are systematically fixed on the form of co-evolution to enhance coverage and pass rate.

Amplification based solutions are also conspicuous. Small-Amp (Panichella et al., 2016) is an extension of Python tests that builds dynamic analysis and the assertion mining based on

heuristics. MuTAP-based mutation-guided methods (Dakhel et al., 2024) introduce new test cases to mutants, which extends test suites with fault-oriented behavior.

These results indicate that test repair and augmentation are becoming inherent attributes of contemporary automated testing devices, such that the generated tests can be run, are reliable and behave as intended. **Table 2-10** shows the repair and augmentation mechanism of each of the tools.

Table 2-10 Test Repair and Test Suite Augmentation Approaches

#	Reference	Input	Tool Developed	Technique	Output	Validation
1	Mani & Attaranasl, 2025	Failing Test Suite	Adaptive Healing System	RL + GPT-4 Test Repair	Repaired Tests	Industrial CI Study
2	Aichernig & Havelund, 2025	Source Code + NL Specification	Nil	Property-Based Refinement	Improved Tests	Case Study
3	Mock et al., 2025	Developer TDD Tasks	Nil	Iterative Test-First LLM Generation	Updated Tests + Code	Practitioner Study
4	Stallenberg et al., 2022	Source Code	GuessWhat	Probabilistic Type Inference Repair	Coverage-Improved JS Tests	Empirical Evaluation
5	Wang et al., 2025	Source Code	HITS	Slice-Based Repair	Coverage-Improved Tests	Empirical Evaluation
6	Pan et al., 2024	Source Code + Mutants	Nil	Mutation-Driven Repair	New Tests	Empirical Evaluation

7	Gu et al., 2024	Source Code	TestART	Co-Evolutionary Repair Loop	Strengthened JUnit Tests	Large-Scale Evaluation
8	Qiu et al., 2024	Source Code	RL Generator	Chain-of-Thought + RL Repair	Improved Tests	Empirical Evaluation
9	Dakhel et al., 2024	Source Code	MuTAP	Mutation-Guided Test Repair	Repaired Python Tests	Benchmark Evaluation
10	Panichella et al., 2016	Source Code + Existing Tests	Small-Amp	Dynamic Test Amplification	Amplified Tests	Empirical Evaluation

2.2.1.3 Tools Evaluation Outcomes

They critically examine the tools and structures identified (as exuded in the technical table and summarized in above tables) along the characteristics set. This assessment aims to gain insight into the level at which current tools assist in the automated unit test generation processes such as test creation, verification, assertion generation, test repair and test augmentation. Additionally, they reviewed whether there were any commonly used research or industrial tools which were absent in the chosen studies. During this analysis, they identified multiple auxiliary tools (e.g., JaCoCo, MutPy, pytest, Mocha, OpenClover) that, while not classified as stand-alone test-generation frameworks, play an important supporting role in verification and execution workflows across numerous studies.

Based on the detailed assessment of the tools in previous tables, they categorize automated testing tools into four primary activity domains:

1. test generation,
2. test verification and improvement,
3. assertion/oracle generation, and
4. test repair and augmentation.

Each category is described in detail below.

i) Test Generation Tools

They identify several tools that primarily support test generation activities. These include both LLM-based frameworks and classical search- or analysis-driven tools.

LLM-driven generators such as ChatUniTest (Xie et al., 2023), TestART (Gu et al., 2024), HITS (Wang et al., 2025), MuTAP (Dakhel et al., 2024), Mutation-Guided Test Generator (Pan et al., 2024), and large-scale prompting evaluations (Ouédraogo et al., 2024) automatically synthesize JUnit or NUnit tests directly from source code, natural-language descriptions, or previously failing tests. These frameworks typically integrate prompt engineering, contextual retrieval, or LLM reasoning workflows to infer test structure and expected behaviors.

Non-LLM tools, such as GuessWhat (Stallenberg et al., 2022), SynTest-JavaScript (Olianas et al., 2022), Nessie (Arteca et al., 2022), and MATTER (Olianas et al., 2022) generate automatically using probabilistic type inference, search-based techniques, symbolic execution, or model-based analysis. These are often based on JavaScript, Python, or IoT-specific environments, and are competitive in controlled evaluation.

A small number of studies explore domain-specific test generation. For example, LLM-based unit test creation for game development (Paduraru et al., 2024) addresses Unity-based game logic, demonstrating how specialized prompts can tailor test generation to graphical and event-driven environments.

The inputs, algorithms, and validation datasets associated with these test generation tools are described in **Table 2-7**.

ii) Test Verification and Improvement Tools

Verification and improvement tools form a significant portion of the identified frameworks and are essential for evaluating the quality of generated tests. Mutation-based verification is the most dominant approach across the selected studies. Tools such as LLMorpheus (Palacios et al., 2024), Google Mutation Service (Petrović et al., 2021), MuTAP (Dakhel et al., 2024),

and the frameworks by Pan et al. (2024) and Liu et al. (2025) rely on mutation scores to assess fault detection and guide the generation or refinement of tests.

Coverage-based verification appears in works such as HITS (Wang et al., 2025) and EvoSuite-integrated evaluations in multiple LLM studies. These tools use coverage metrics to iteratively improve test completeness or to detect untested behavior slices.

Differential and behavioral verification tools, such as JEST (Park et al., 2021) and DiffGen (Barr et al., 2014), validate behavioral correctness by executing test cases across program versions or runtime engines. They are very customized tools, which are efficient even when dealing with complex languages like JavaScript.

It is a common observation that verification tools are considered to be closely woven with test-generation pipelines. As an example, mutation-guided generation of LLMs and co-evolutionary feedback loops are generation and verification, respectively, and the holistic nature of current automated testing processes.The main verification and improvement tools, with their outputs and datasets, are summarized in **Table 2-8**.

iii) Assertion and Oracle Generation Tools

Assertion generation tools form a specialized category focused on improving oracle quality. Tools such as AsserT5 (Primbs et al., 2025), ChIRAAG (Pahwa et al., 2024), and AugmenTest (Khandaker et al., 2025) apply LLM reasoning, masked language modeling, or retrieval-based inference to produce assertions tailored to the tested method's semantics.

Hybrid IR+DL approaches (Yu et al., 2022) extract candidate assertions from historical test repositories and refine them to fit the target context. These techniques demonstrate that oracle correctness remains one of the most challenging components of automated test generation.

They find that certain assertion-generation tools can also assist in the repair or strengthening of tests, partway as other property-specification tools in other fields can assist in modeling and verification. Such tools are thus grouped under the assertion generation where oracle refinement is the main activity. **Table 2-9** has their characteristics and assessment settings.

iv) Test Repair and Augmentation Tools

Test repair and augmentation tools are found in various studies as independent processes to improve on the failure of tests, poor oracle, or poor coverage. Solutions based on reinforcement-learning, including Adaptive Test Healing (Mani and Attaranasl, 2025) and RL-guided generation (Qiu et al., 2024), use the feedback of the execution to select the corrective actions that generate the stable passing tests.

Co-evolutionary strategies (Gu et al., 2024) integrate template-based repair with iterative refinement, yielding progressively improved JUnit suites. Test augmentation tools such as Small-Amp (Panichella et al., 2016) add new input scenarios, expand coverage, or generate additional behavior-based tests using dynamic instrumentation.

These repair and augmentation tools highlight the emerging trend that automated test generation is no longer limited to initial test creation but increasingly supports full lifecycle maintenance of the generated test suites.

2.2.1.4 Tools Efficiency, Maturity, and Compatibility

They evaluate tools according to their level of validation, implementation maturity, execution environment, and integration capability with existing testing ecosystems.

There are numerous LLM-based systems, including ChatUniTest, TestART, HITS, and MuTAP, which are highly mature and tested on standard datasets, e.g., Defects4J, SF110, HumanEval, or even large collections of open-source repositories. Their products are compatible with the existing testing systems like JUnit, pytest, NUnit, and Mocha, and they blend with the current CI/CD systems.

Non-LLM tools, including GuessWhat, SynTest-JS, MATTER and Nessie, have also been widely tested on real-world OSS projects, and showed real-world reliability and high scalability.

Other researches only provide some initial prototypes or descriptions of the tools. They comprise initial TDD research (Mock et al., 2025; Bernal-Cardenas et al., 2023) and the idea of assertion-generation models. The fact that little details are presented on how these tools are

implemented does not, however, detract on the overall utility of the tools in identifying emerging research trends.

They also observe that many tools rely on supportive verification utilities (e.g., JaCoCo, MutPy, MuJava, OpenClover, ScalaCheck) which, although not stand-alone test generators, play critical roles in measurement and validation.

Overall, the tools included in this SLR present a diverse ecosystem of LLM-based and traditional capabilities that collectively advance the state of automated unit test generation.

2.2.2 Answers of Research Questions

Research Question 1:

What important researches have been reported from 2018 to 2025 where automated test generation techniques (LLM-based and non-LLM-based) have been utilized?

Answer:

A total of **48 important research works**, published between **2018 and 2025**, have been identified according to the selection and rejection criteria. These studies have been classified into five predefined categories. The details are as follows:

- **Ten studies** have been located under **NLP Techniques** category. They use semantic parsing, methods based on retrieval, prompt engineering, or learned textual models to facilitate the many parts of test generation.
- The **Verification & Improvement** category has identified **thirteen researches**. These contributions focus on mutation testing, coverage optimization, differential execution and dynamic repair loops.
- In **Tools & Frameworks** category, **fourteen** researches are found. These are end-to-end test generation, assertion enhancing, code slicing and coverage frameworks.

- **Nine researches** have been identified in the **Algorithms & Transformation Techniques** category. These include evolutionary algorithms, reinforcement learning, type-inference algorithms, slicing algorithms, and symbolic-execution-driven strategies.
- **Six researches** have been identified in the **Research Gaps & Future Directions** category. These focus on limitations in LLM-generated tests, oracle quality issues, coverage gaps, robustness concerns, and open research challenges.

The overview, validation method, inputs, outputs, and algorithmic details of each selected work are provided in the technical table.

Research Question 2:

What NLP techniques have been applied for automated test generation and software testing?

Answer:

It has been analyzed that NLP techniques are widely utilized across multiple studies, not only as standalone components but also integrated with LLM-based generation methods. Based on the selected works:

- NLP techniques are used for **semantic code understanding**, **intent extraction**, **retrieval of relevant assertions**, **text-to-code conversion**, and **prompt-driven reasoning**.
- Studies such as (Taromirad & Runeson, 2025; Yu et al., 2022; Jiang et al., 2024) rely on linguistic and semantic analysis to infer expected behavior.
- Prompt engineering (zero-shot, few-shot, chain-of-thought, tree-of-thought, property-encoding prompts) is extensively used in LLM-based test generation frameworks.
- Retrieval and similarity-based NLP techniques are used to locate likely assertions from prior repositories.

Research Question 3:

What tools have been developed or reused for automated unit test generation, and what inputs and outputs do these tools support?

Answer:

Based on the SLR, **14 primary tools/frameworks** have been identified to support automated test generation, test repair, assertion inference, or reinforcement-learning-based refinement. These tools are reported in the technical table and summarized as follows:

- Tools supporting **test generation** include: ChatUniTest, HITS, TestART, MuTAP, GuessWhat, SynTest-JS, Nessie, and MATTER.
- Tools supporting **verification and validation** include: LLMorpheus, Google Mutation Service, JEST, DiffGen, and mutation frameworks embedded in LLM-based tools.
- Tools supporting **assertion generation** include: AsserT5, ChIRAAG, AugmenTest, and IR+DL assertion generators.
- Tools supporting **test repair or augmentation** include: Adaptive Test Healing, TestART, Small-Amp, and RL-guided repair processes.

Each tool has been analyzed according to input type (e.g., Java methods, JS functions, mutants, logs), underlying algorithm (e.g., reinforcement learning, slicing, evolutionary methods), and final outputs (JUnit tests, assertions, repaired tests)..

Research Question 4:

What algorithms have been developed or applied in automated unit test generation, and how do these algorithms transform inputs into testing outputs?

Answer:

A total of **nine significant algorithmic approaches** has been identified. These algorithms transform source code, mutants, execution traces, or natural-language specifications into test cases or repaired test suites. The analysis reveals:

- **Evolutionary algorithms** (e.g., NSGA-II, SMSEMOA) are used for generating optimized test suites (Abdallah et al., 2018).
- **Reinforcement learning algorithms** are used for test repair, flaky-test stabilization, and refinement strategies (Mani & Attaranasl, 2025; Qiu et al., 2024).
- **The slicing algorithms** are applied to obtain targeted prompts and enhance coverage (Wang et al., 2025).
- JavaScript test generation is assisted by **probabilistic type-inference algorithms** (Stallenberg et al., 2022).
- Guided by **mutation algorithms** using mutations to guide the generation of running tests.
- The algorithms **Hybrid IR + deep learning retrieve** and modify assertions (Yu et al., 2022).
- Stub repair is supported by the **constraint-extraction algorithms** (Zhang et al., 2023).

These algorithms constitute the fundamental transformation methods that encode program semantics into structured unit tests, assertions, repair actions or behavioral checks.

The gaps in these research point to the possibility of improving automated unit test generation by using hybrid methods, multi-model reasoning, stronger oracle construction and standardized experimental protocols.

2.3 Discussion and Limitations

This section presents the general results of the systematic literature review and reveals their importance in the domain of automated unit test generation. It compares the advantages and limitations of the current solutions, such as the old methods, NLP-based methods, and the new LLLM-based methods. The discussion identifies repetitive problems including weak assertion, weak verification, inconsistent mutation, difficulty in dependency-handling and weak real-world validation. The constraints that have been identified in the past research works assist in exposing some significant research gaps in the literature and gives a clear starting point in the definition of the research objectives of this thesis.

2.3.1 Discussion on NLP Techniques for Automated Test Generation

It is discussed that NLP-based approaches have become a key part of the current test generation methods by automated means. NLP facilitates the language of nature descriptions, comments in the code, semantics of their methods and expected behavior. NLP is not a standalone mechanism used in the majority of the chosen researches, but in combination with other techniques, which may be mutation testing, repair loops, or coverage guidance. To illustrate this, the research conducted by (Taromirad, Runeson, 2025) and (Yu, et al., 2022) and (Jiang, et al., 2024) show that semantic understanding and similarity retrieval are highly beneficial in making meaningful assertions and test structure.

Nevertheless, it is still hard to combine the NLP techniques with the static or dynamic analysis. Ambiguity in natural-language descriptions, inability to translate text-based requirements into accurate oracles, and the possibility of LLM hallucinations are some of the issues noted in some works. The rapid engineering methods vary greatly among the different tools, and some methods involve domain-specific templates to produce consistent results. Nevertheless, the challenges notwithstanding, NLP techniques have received significant adoption due to the fact that they facilitate expressive reasoning and also minimize the manual efforts needed to process the functional requirements.

2.3.2 Discussion on Test Generation Approaches

Test generation is the most prevalent activity in most of the studies selected as it is modeling in other engineering disciplines. The developed test generation frameworks, such as ChatUniTest, HITS, TestART, Mutation-Guided Test Generation, and MuTAP, based on the work of LLM, have been shown to be quite effective at generating readable and structurally valid unit tests. The tools make use of the contextual prompts, slicing, retrieval augmentation or mutation-based cues to produce tests that are reminiscent of developer written code.

Non-LLM approaches, on the other hand, retain heavy abilities of test generations through search based optimization, type inferences, symbolic execution, and model based reasoning.

GuessWhat, SynTest-JS and Nessie confirm that older techniques remain competitive, particularly in dynamically typed languages such as JavaScript and Python, where scaffolding of the techniques based on the use of LLM is often required.

Among the primary observations, it can be stated that pure test-generation techniques are becoming rare. Most of the tools involve a combination of several downstream processes, like coverage testing, repairing failed test cases, test hardening, or test suite expansion, implying that they have transitioned to end-to-end pipelines and not a single test generator.

2.3.3 Discussion on Test Verification and Quality Optimization

Test verification is a very important stage of the correctness and stability of the automatically generated tests. Another important stream of research is based on mutation testing to test the effectiveness of tests (Liu et al., 2025; Pan et al., 2024; Palacios et al., 2024; Dakhel et al., 2024). Not only do the mutation-guided strategies assess the adequacy of the tests, but also offer the feedback that propagates cycles of refinements.

Enhancement of coverage is also highly utilized. The application of method slicing and coverage analysis to identify untested segments of code and guide further test generation are used in such tools as HITS (Wang et al., 2025). Good behavioral testing can be provided to multi-runtime or versioned systems with differentiable testing systems, such as JEST (Park et al., 2021) and DiffGen (Barr et al., 2014).

Even though the ways of verification are most effective in enhancing reliability, it is still inadequate in some situations. Tests generated by the LLM are usually very readable but their mutation-kill rates are not consistent. In addition, mutation testings have a heavy computation cost, and large industrial codebases are difficult to scalable to the frameworks. Even coverage analysis does not ensure semantic correctness and the only application of differential testing is in the situation where there are multiple programs or versions of programs with similar run time.

2.3.4 Discussion on Assertion and Oracle Generation

The most challenging part of automated test generation is assertion generation. AsserT5, ChIRAAG, and AugmenTest offer the means to draw or narrow assertions with the help of LLMs, IR-based retrieval, or masked language models. These tools are very helpful in enhancing oracle, but there are still limitations.

The statements made by LLMs can be syntactically valid but semantically poor. In most instances, there is trivial behavior in generated assertions, are based on over-specificity in the expected values, or have corner cases not represented. The retrieval-based methods involve having rich test repositories and the deep-learning methods are very sensitive to the quality of training-data and need to cover the domain.

An interesting point to note is that assertion-generation systems tend not to be independent primary test-generators, but instead to be in combination with primary test-generators. It means that there is an increasing awareness that to build reliable oracles, dedicated methods are needed that would allow oracle building based solely on general-purpose test generation.

2.3.5 Discussion on Test Repair and Test Augmentation

Some of the recent studies give test repair and augmentation activities prominence. The objectives of these mechanisms are to fix flaky tests, runtime failures, unavailable assertions or lack of coverage.

Reinforcement-learning-based repair mechanisms (such as Adaptive Test Healing (Mani and Attaranasl, 2025) and RL-guided refinement (Qiu et al., 2024)) make a selection of corrective actions based on runtime performance. The TestART co-evolutionary methods (Gu et al., 2024) involve the repetitive process involving repairing and regenerating tests and improving the pass and the percentage of test coverage.

Previous tests can be augmented with novel inputs or new behaviors are maximized by amplification methods such as the Small-Amp (Panichella et al., 2016). These strategies indicate a growing trend on keeping and upgrading test suites during software lifecycle.

Nevertheless, most repair frameworks need large amounts of execution feedback and use language-specific tooling or mutation environments, and thus cannot be applied to multi-language or large-scale industrial contexts.

2.3.6 Discussion on Tool Selection and Compatibility

The use of automated unit test generation requires the selection of the right tools because there is a great variety of testing objectives, language eco systems, and infrastructure demands. There are major differences in the tools in the way they support programming languages, can be used with different frameworks, are based on mutation engines, or based on LLMs.

Tools based on LLM will frequently need access to strong language models, and their results can be different across models or promptings. Manual configuration of a type inference, symbolic execution engine, or coverage instrumentation can be needed with traditional tools, although such tools can be more predictable.

The other challenge is compatibility. A wide variety of frameworks are based on testing libraries e.g. JUnit, pytest, NUnit, or Mocha. Mutation engineers integrated as tools (e.g., MutPy, MuJava, OpenClover) have to be compatible with that language version, and compatible with that build environment, making their use in polyglot or cloud-based development systems difficult.

Nevertheless, most of the found tools have a high potential of real-world use, especially those that were tested on Defects4J, SF110, HumanEval, or giant JavaScript/Python repositories.

2.3.7 Limitations of Research

Though this SLR complies with accepted methodology of systematic reviews and acts within the determined review protocol, there are a number of limitations:

- **Search space limitations:** They searched with well thought search terms and browsed through retrieved results. Nevertheless, certain terms in search yielded thousands of

results making it impossible to scan through them manually. It is conceivable that few of the potentially relevant studies might have been left out owing to their titles or abstracts not clearly indicating the entire content.

- **Database selection:** This SLR is based on four renowned scientific repositories: IEEE, ACM, Springer, and Elsevier. Although these databases provide high-quality literature and dominate software engineering research, relevant works from additional databases (e.g., arXiv, Scopus, or domain-specific archives) may not have been captured. Nevertheless, given the authority and coverage of the selected databases, they believe the overall findings are not significantly affected.
- **Tool documentation variability:** Several studies do not fully describe internal tool implementations, configurations, or execution environments. As a result, tool comparison is based on available descriptions, which may vary in depth and completeness.
- **Rapid evolution of LLM-based approaches:** Because LLM-based test generation is evolving rapidly, new techniques and models may emerge after the completion of this SLR. Consequently, the findings represent the state of research up to the final search date but may require future updates.
- **Database coverage limitations:** They have accessed four well-known scientific databases i.e., **IEEE, Elsevier, ACM** and **Springer** as the main sources of finding the relevant studies. These databases are peer-reviewed sources of journal and conference publications in software engineering and artificial intelligence of great volume. But other repositories like arXiv, Scopus and domain specific archives also have supplementary research in them. Thus, it is possible that not all the recent or developing works, especially the preprints or reports in the industry were not covered in this review. However, they feel that no significant difference is caused on the overall findings of this SLR, as the databases that have been used are the most authoritative and more commonly used venues of research on automated unit test generation and LLM-based testing.

2.4 Conclusion

This study describes the current trends, methods, algorithms, and tools that assist in automated generation of unit tests based on LLM-based and conventional methods. To achieve this goal, a Systematic Literature Review (SLR) was performed to select **48 research works** that are relevant to the study, published in 2018-2025. These studies were further grouped into **five categories** depending on pre-determined categories that include NLP techniques, verification and improvement methods, automated testing tools, algorithmic transformation methods and gaps in research.

After this, the identified tools and techniques were analyzed thoroughly with respect to different significant features that are considered when it comes to automated unit test generation. This discussion led to classification of tools into four broad testing activities namely **test generation**, **test verification and improvement**, **assertion** and **oracle generation** and **test repair and augmentation**. All groups have their own unique practices, capabilities and challenges of enhancing the quality and reliability of automatically generated test suites.

This study also examines the application of different NLP algorithms such as prompt engineering, semantic parsing, retrieval-based inference and hybrid learning models besides tool evaluation. These techniques are combined with reinforcement learning, mutation testing, slicing algorithms, and search-based methods in a trend that is gradually moving towards the use of hybridized techniques in automated testing. Also, other forms of algorithmic strategies like multi-objective evolutionary optimization, probabilistic type inference, and coverage-guided refinement have been examined in terms of their ability to increase test correctness and structural adequacy.

In this way, the current trends, methods, and valuable tools of automated unit test generation, both based on the paradigms of LLM and classical paradigms of search-based (and analysis-based) paradigms are summarized within one extensive study. Such coherent overview is, to the best of our knowledge, insufficient in literature. The results of this SLR will help the researcher, practitioner, and tool developer to choose the right techniques, tools, and

verification strategy based on their testing needs and also indicate the new possibilities of developing automated test generation.

Chapter 3

3. Requirements Specification and Analysis

The purpose of this chapter is to define and analyze the requirements of the proposed system, *A Unified Framework for Verification & Improvement of LLM-Based Automated Unit Test Generation.* This chapter outlines the functional and non-functional requirements through Epics, User Stories, Test Cases, and User Interface design. It also provides a Traceability Matrix to ensure all requirements are connected with appropriate user stories, test cases, and UI components.

The system is built as a **VS Code Extension** integrated with a unified backend framework that detects, verifies, repairs, and improves LLM-generated unit tests. The system supports JavaScript/Node.js testing workflows and integrates verification methods such as mocking, adaptive repair loops, assertion strengthening, and mutation testing.

3.1 Epics

3.1.1 Epic E1: Upload Source Code

Epic ID: E1

Description:

The user shall be able to give the system a JavaScript/Node.js project as input. This connects the tool to a real development workflow.

3.1.2 Epic E2: LLM-Based Unit Test Generation

Epic ID: E2

Description:

The system shall use an LLM (e.g., GPT-3.5) to generate initial unit tests for selected functions or modules in the project, using the available code and metadata.

3.1.3 Epic E3: Dependency Detection and Mock Generation

Epic ID: E3

Description:

The system shall detect external dependencies (APIs, database calls, file system, environment variables, network calls) and automatically generate mocks or stubs. This makes LLM-generated tests executable even when external systems are not available.

3.1.4 Epic E4: Adaptive Repair Loop

Epic ID: E4

Description:

When a test fails, the system shall automatically start a repair cycle: classify the failure, explain the reason to the LLM, update the test, and re-run it. This is repeated until the test passes or a limit is reached.

3.1.5 Epic E5: Assertion Quality Enhancement

Epic ID: E5

Description:

The system shall detect weak or trivial assertions and suggest or generate stronger ones using LLMs and runtime data, so that tests can actually catch real bugs.

3.1.6 Epic E6: Developer Interface (VS Code UI)

Epic ID: E6

Description:
The system shall provide a clear and friendly UI inside VS Code (and optionally web) where the user can upload/select the project, run the pipeline, see failing tests, repair suggestions, mocking suggestions, mutation score, coverage, and before/after comparisons.
At the end of the pipeline, the final generated and improved test suite shall be available for download.

3.2 User Stories

Now, each Epic is broken into **user stories** in the standard format:

3.2.1 User Stories for Epic E1: Upload Source Code

E1-US1: Source Code Selection / Upload

Role: User (Developer)

Description:
A JavaScript/Node.js source code shall be selected or uploaded in the UnitGen interface so that the system knows which source code it shall analyze and generate tests.

Acceptance Criteria:

- The interface allows the user to pick a local project folder or upload a zipped Node.js project.
- The system validates that the project contains JavaScript/TypeScript source files.
- If the project is invalid, a clear error message is shown.

3.2.2 User Stories for Epic E2: LLM-Based Unit Test Generation

E2-US1: Generate Initial Unit Tests with LLM

Role: System

Description:
Initial Unit Tests shall be generated using an LLM for the selected functions or modules so that the developer gets a starting test suite quickly.

Acceptance Criteria:

- The user can select one or more functions/modules from the project.
- The system sends structured context (code, signatures, comments) to the LLM.
- The generated tests are valid code files and can be executed by the runner.

3.2.3 User Stories for Epic E3: Dependency Detection and Mock Generation

E3-US1: Detect External Dependencies

Role: System

Description:
External dependencies such as file-system calls, HTTP requests, databases, and environment variables shall be detected across the project so that they can be mocked during test execution.

Acceptance Criteria:

- The system lists each detected dependency with file name and line number.
- Types of dependencies (API, DB, file system, env, network) are clearly labeled.

E3-US2: Generate Mock Templates

Role: System

Description:

Mock or stub templates (using tools like Sinon or Nock) shall be generated for each detected dependency so that tests can run without real external systems.

Acceptance Criteria:

- For each dependency, a suggested mock code snippet is created.
- Mock snippets can be viewed and edited in the UI.

E3-US3: Apply Mocks Automatically

Role: System

Description:

Selected mocks shall be automatically applied at test runtime so that tests execute in isolation and do not call real external services.

Acceptance Criteria:

- When tests run through UnitGen, real external calls are intercepted.
- Logs show that mocks are used instead of real calls.

3.2.4 User Stories for Epic E4: Adaptive Repair Loop

E4-US1: Classify Failure Reason

Role: System

Description:

Each Failing Test shall be classified based on its failure reason (syntax error, runtime error, dependency problem, assertion failure, timeout) so that the correct repair strategy can be chosen.

Acceptance Criteria:

- Each failing test is assigned at least one failure category.

- Categories are visible in the UI.

E4-US2: Repair and Re-Run Failed Tests

Role: System

Description:
Failure Details shall be sent to the LLM, update the test, and re-run it so that failing tests can be automatically repaired.

Acceptance Criteria:

- For a failing test, the system generates a repaired version.
- The repaired test is re-executed.
- The system stops either when the test passes or when a configured max number of repair attempts is reached.
- A small repair history (before/after) is kept.

3.2.5 User Stories for Epic E5: Assertion Quality Enhancement

E5-US1: Detect Weak Assertions

Role: System

Description:
Tests containing weak or trivial assertions (for example, always-true checks) so that they can be improved.

Acceptance Criteria:

- Trivial or constant comparisons are flagged.
- Flagged tests are listed in the UI for review.

E5-US2: Suggest Stronger Assertions

Role: System

Description:

Stronger assertions shall be suggested using runtime values and function behaviour so that tests actually validate meaningful outcomes.

Acceptance Criteria:

- For a flagged weak assertion, the system can propose one or more stronger assertions.
- The developer can accept or reject suggestions.

3.2.6 User Stories for Epic E6: Developer Interface (VS Code UI)

E6-US1: Main Control Panel

Role: User (Developer)

Description:

A main VS Code control panel shall be displayed where the project can be selected, the pipeline can be started, and UnitGen features can be accessed from one location.

Acceptance Criteria:

- The extension shows a clear header and project input bar.
- Buttons to “Generate Tests”, “Run Full Pipeline”, and “Download Suite” are visible.

E6-US2: Results and Feedback View

Role: User (Developer)

Description:

Test results fail or pass and other metrices shall be shown in a readable panel so that the tool’s actions can be clearly understood.

Acceptance Criteria:

- Failed and passed tests are listed.
- Repair and assertion suggestions can be viewed side-by-side (before/after).

E6-US3: Download Final Test Suite

Role: User (Developer)

Description:
The complete set of generated and improved tests shall be provided as a downloadable zip file.

Acceptance Criteria:

- After the pipeline finishes, a "Download Test Suite" button becomes available.
- The downloaded zip contains all relevant test files in a standard folder structure.
- The download includes a simple README describing how to run the tests.

3.3 Representative Test Cases

Below are representative test cases aligned with the user stories:

3.3.1 Test Case 1

Table 3-1 TC1: Source Code Upload and Validation

Test ID: E1-US1-TC1	**US ID: E1-US1**
Test Case Description:	Verify that the user can upload valid Node.js source code and that invalid source code is rejected with a clear error message.
Scenario:	User uploads valid or invalid Node.js source code and the system validate the structure.

Inputs:	Valid Node.js source code folder or zip containing package.json and JavaScript files. Invalid source code folder containing no JavaScript files.
Expected Result:	For valid source code, the system accepts the input and confirms readiness. For invalid source code, the system shows an error message stating that no JavaScript or Node.js source files were found.

3.3.2 Test Case 2

Table 3-2 TC2: Detection of External Dependencies

Test ID: E3-US1-TC1	**US ID: E3-US1**
Test Case Description:	Verify that the system detects external dependencies inside the project code.
Scenario:	User uploads a project and initiates dependency analysis.
Inputs:	A project that uses fs.readFileSync and axios.get.
Expected Result:	The system lists detected dependencies including fs.readFileSync and axios.get with file and line number. Each dependency is labeled as file system or HTTP.

3.3.3 Test Case 3

Table 3-3 TC3: Mock Template Generation

Test ID: E3-US2-TC1	**US ID: E3-US2**
Test Case Description:	Verify that mock templates are generated for all detected dependencies.
Scenario:	System receives dependency list and generates corresponding mock templates.
Inputs:	Dependency list obtained from TC2.

Expected Result:	A Sinon stub template is created for fs.readFileSync. A Nock template is created for axios.get.

3.3.4 Test Case 4

Table 3-4 TC4: LLM Based Unit Test Generation

Test ID: E2-US1-TC1	**US ID: E2-US1**
Test Case Description:	Verify that the system generates executable unit tests uaing the LLM
Scenario:	User Selects a function and requests automatic test geenrations.
Inputs:	A javascript function with JSDoc comments and simple logic such as add a comma b.
Expected Result:	A structured prompt is sent to the LLM. At least one valid test file is created. The test runs without syntax errors.

3.3.5 Test Case 5

Table 3-5 TC5: Failure Classification and Repair

Test ID: E4-US1-TC1	**US ID: E4-US1 and E4-US2**
Test Case Description:	Verify that failures are classified and repairs are attempted.
Scenario:	System runs a test suite containing multiple failure types and attempts to repair recoverable issues.
Inputs:	One test containing a syntax error. One test failing due to missing file. One test failing due to incorrect expected value.

Expected Result:	Failures are classified as syntax error, dependency error and assertion failure. Dependency and assertion failures are repaired and re-executed. Results are shown to the user.

3.3.6 Test Case 6

Table 3-6 TC6: Weak Assertion Detection and Improvement

Test ID: E5-US1-TC1	**US ID: E5-US1 and E5-US2**
Test Case Description:	Verify that weak assertions are detected and improved alternatives are suggested.
Scenario:	System analyzes assertions inside test files and flags trivial ones.
Inputs:	A test file containing expect true toBe true and assert equal one comma one and a meaningful assertion such as expect add two comma three toBe five.
Expected Result:	Weak assertions are flagged. The meaningful assertion remains unflagged. At least one stronger suggestion is generated.

3.3.7 Test Case 7

Table 3-7 TC7: Download Final Test Suite

Test ID: E-US3-TC1	**US ID: E6-US3**
Test Case Description:	Verify that the final test suite can be downloaded.
Scenario:	User completes the pipeline and downloads the generated test suite.
Inputs:	A completed pipeline run.
Expected Result:	Selecting the download option triggers a zip download. The zip contains all final test files in correct folder structure.

3.4 Traceability Matrix:

Table 3-8 Traceability Matrix

Epic	User-Stories	Test-Cases	UI Components
E1: Upload Source Code	E1-US1	E1-US1-TC1	UI-1: Main UnitGen Interface (Header + File Input Bar + Generate Button)
E2: LLM-Based Unit Test Generation	E2-US1	E2-US1-TC1	UI-1: Main Interface (“Generate Tests” action performs background LLM test generation)
E3: Dependency Detection and Mock Generation	E3-US1, E3-US2, E3-US3	E3-US1-TC1, E3-US2-TC1	UI-1: Same Main Interface (dependency analysis + mock generation done in background)
E4: Adaptive Repair Loop	E4-US1, E4-US2	E4-US1-TC1	UI-2: Output Panel (shows repaired test results, before/after text, success status)
E5: Assertion Quality Enhancement	E5-US1, E6-US2	E5-US1-TC1	UI-2: Output Panel (shows weak assertion warnings + improved suggestions)
E6: Developer Interface (VS Code UI)	E6-US1, E6-US2, E6-US3	E6-US3-TC1	UI-1 + UI-2: Entire interface displayed inside VS Code Webview and Download Zip Button

3.5 Basic Interface Design:

This design represents the basic UI layout that will be used as part of the UnitGen VS Code Extension. The primary goal of this interface is to provide a clean, intuitive, and developer-friendly environment for interacting with the test generation and improvement features offered by the extension.

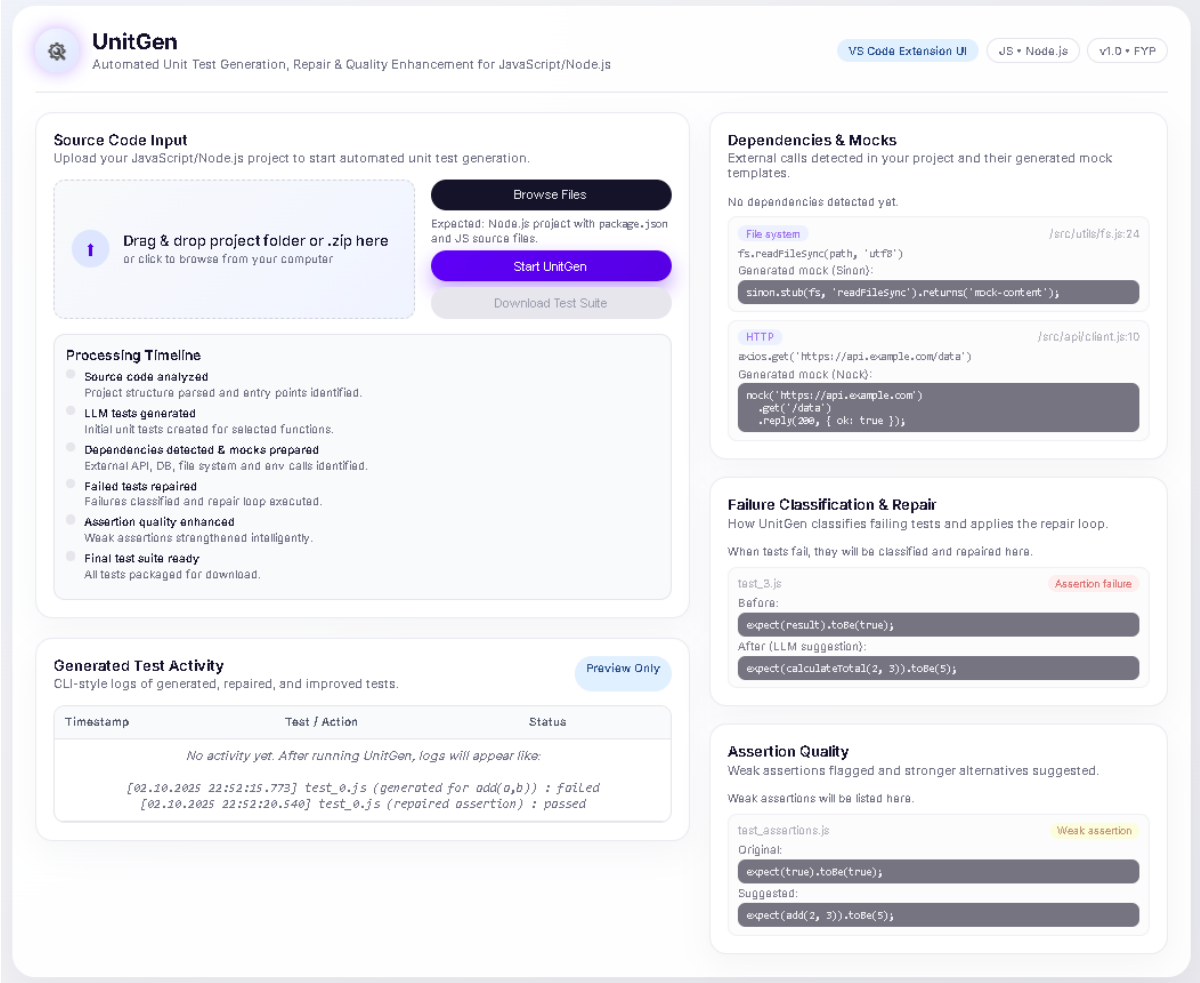

Figure 3-1 Graphical User Interface

Chapter 4

4. Proposed Solution

This chapter presents the proposed solution for addressing the limitations of existing automated and LLM-based unit test generation approaches identified in the earlier chapters. While recent Large Language Models (LLMs) are capable of producing readable and syntactically correct unit tests, their practical usability is often limited due to missing external dependencies, weak assertions, lack of verification, and poor integration with real development workflows.

To overcome these challenges, this project proposes a unified framework that not only generates unit tests using LLMs but also verifies, repairs, and improves them through a structured and iterative pipeline. The solution integrates static code analysis, automatic dependency mocking, LLM-guided test synthesis, execution-based repair, and quality enhancement mechanisms into a single cohesive system.

The proposed framework operates at the function level, enabling precise and isolated unit test generation. It begins by statically analyzing the source code to extract individual functions and their external dependencies. Based on this analysis, executable test templates and mocks are automatically generated. These templates are then completed using an LLM to produce initial unit tests.

Unlike existing tools that stop after test generation, the proposed solution introduces a verification-driven workflow. Generated tests are executed, and any failures are handled using an adaptive repair loop that leverages execution feedback to iteratively refine the tests. Furthermore, assertion quality is enhanced and mutation testing is applied to assess and improve the fault-detection capability of the generated test suites.

The framework is designed with modularity and extensibility in mind and supports both command-line and integrated development environment (IDE) usage. By combining generation, verification, and improvement into a single pipeline, the proposed solution bridges

the gap between research-oriented test generation and practical, developer-usable automated testing.

The following sections detail the software architecture, data modeling approach, system workflow, and third-party dependencies that collectively realize this proposed solution.

4.1 Software Architecture

The proposed solution is designed as a **layered and modular architecture** that integrates static analysis, automated mocking, LLM-based test generation, and verification-driven improvement into a unified pipeline. The high-level architecture of the system is illustrated in **Figure 4-1**.

The architecture is divided into the following major layers:

4.1.1. Developer Interface Layer

This layer provides the entry point for developers to interact with the system. It supports:

- A **Command Line Interface (CLI)**, which is currently implemented and allows users to provide source files or project paths.
- A **VS Code Extension**, planned for later phases, which will allow interactive test generation and review directly within the development environment.

This separation ensures that user interaction remains independent of the core system logic.

4.1.2. Core Analysis Layer

The Core Analysis Layer is responsible for **static code inspection**. It includes:

- An **API Explorer**, which discovers source files and modules within the selected project.
- A **Static Code Analyzer (AST-based)**, which parses source files into an Abstract Syntax Tree (AST) to extract individual functions, their parameters, return behavior, and asynchronous properties.

This layer enables precise, function-level understanding without executing the code.

4.1.3. Dependency and Mocking Layer

This layer addresses the challenge of external dependencies in automated test generation. It consists of:

- A **Dependency Detector**, which identifies external libraries, APIs, and system calls used by each function.
- An **Automatic Mock Generator**, which creates appropriate mocks or stubs for detected dependencies.

By isolating unit tests from real external resources, this layer significantly improves test executability and reliability.

4.1.4. Test Generation and Verification Layer

This layer integrates Large Language Models with execution feedback:

- The **Prompt Builder** constructs structured prompts containing function context, dependency information, and test templates.
- The **LLM Test Generator** produces unit test logic by completing these templates.
- The **Test Execution Engine** executes generated tests using a testing framework.
- The **Adaptive Repair Loop** analyzes failures and iteratively refines tests using LLM-guided regeneration.

This iterative mechanism ensures that generated tests converge toward correctness.

4.1.5. Quality and Output Layer

The final layer focuses on improving test effectiveness:

- The **Assertion Quality Analyzer** detects weak or trivial assertions and strengthens them.
- The **Mutation Testing Module** evaluates fault-detection capability.
- The **Test Suite Exporter and Reporting Module** produces finalized tests and quality reports.

This layer ensures that tests are not only runnable but also meaningful and fault-sensitive.

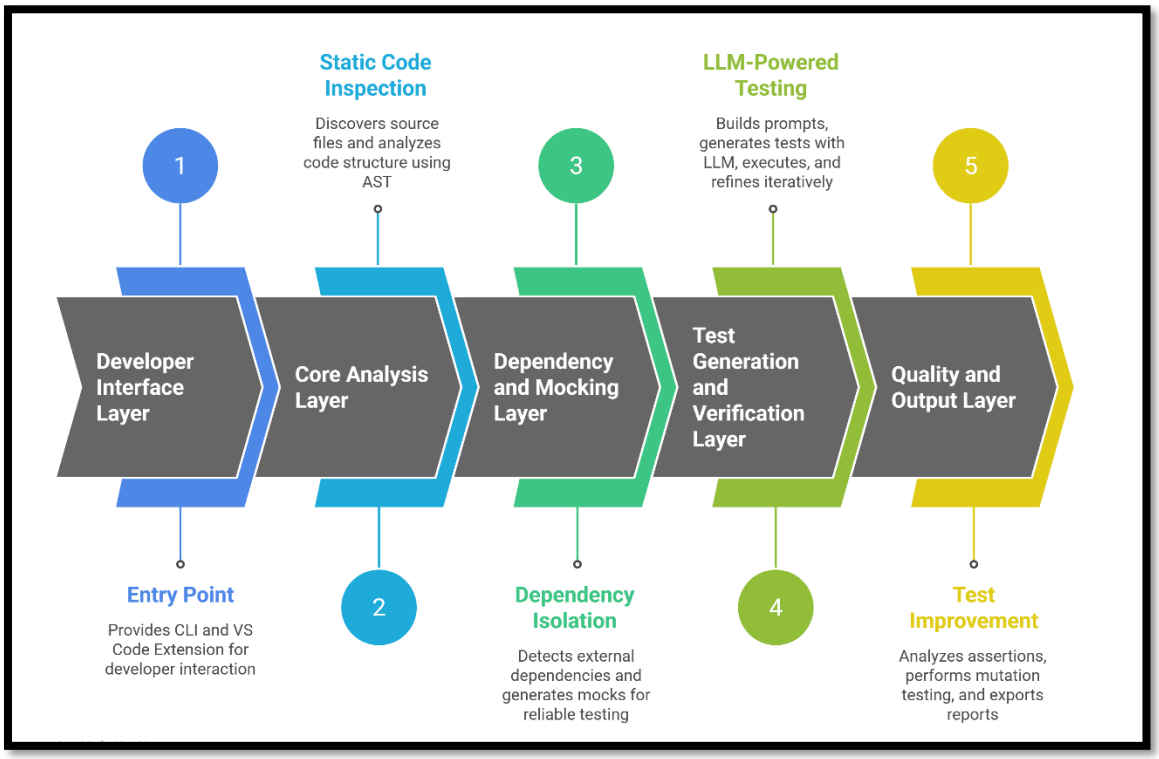

Figure 4-1 Software Architecture Diagram

4.2 Data Modeling

The proposed system adopts a **NoSQL / JSON-based data modeling approach** to represent internal artifacts generated during the execution of the automated test generation pipeline. Unlike traditional software systems that require persistent relational storage, this framework operates as a **pipeline-driven analysis and generation system**, where most data is transient, hierarchical, and execution-specific.

The data model is therefore designed to capture **logical entities and runtime states** rather than enforce rigid relational constraints. This approach improves flexibility, scalability, and alignment with the dynamic nature of JavaScript-based systems and Large Language Model (LLM) workflows. The conceptual NoSQL data model of the system is illustrated in **Figure 4-2**.

4.2.1 NoSQL / JSON-Based Data Model

The NoSQL-based conceptual data model represents system artifacts as nested JSON objects, allowing each pipeline stage to produce and consume structured data without the overhead of schema enforcement or complex joins. This design is particularly suitable for representing evolving artifacts such as extracted functions, generated tests, and quality metrics.

The key entities in the data model are described below.

i) Project

The **Project** entity represents the input codebase or source file set provided by the developer. It serves as the root entity for all subsequent pipeline executions.

Attributes:

- projectId: Unique identifier for the project
- name: Project name
- rootPath: File system path of the source code
- type: Input type (single file or full package)

A single project may undergo multiple analysis and generation executions over time.

ii) PipelineRun

The **PipelineRun** entity represents a single execution instance of the framework. Each run corresponds to one complete traversal of the pipeline, from static analysis to test generation and verification.

Attributes:

- runId: Unique identifier for the pipeline run
- startTime: Execution start timestamp
- endTime: Execution completion timestamp
- status: Overall execution status (e.g., success, partial, failed)

This entity enables tracking, evaluation, and comparison of multiple runs on the same project.

iii) FunctionTarget

The **FunctionTarget** entity represents an individual function extracted from the source code during AST-based static analysis. Each function serves as a primary unit for test generation.

Attributes:

- functionName: Name of the function
- filePath: Source file location
- parameters: List of function parameters
- asyncFlag: Indicates whether the function is asynchronous

A single pipeline run may contain **multiple function targets**, enabling function-level isolation and precise test synthesis.

iv) GeneratedTest

The **GeneratedTest** entity represents a unit test generated for a specific function. This entity captures both structural and execution-related information about the test.

Attributes:

- testFilePath: Location of the generated test file
- framework: Testing framework used (Jest or Mocha)
- content: Generated test code
- executionStatus: Result of test execution (pass, fail, repaired)

Each function target may have **one or more generated tests**, particularly when regeneration occurs during the adaptive repair process.

v) QualityMetrics

The **QualityMetrics** entity captures quantitative measures used to evaluate and improve test effectiveness. This entity is optional and may be populated only after verification stages.

Attributes:

- assertionStrength: Measure of assertion robustness
- mutationScore: Percentage of killed mutants
- repairAttempts: Number of repair iterations performed

These metrics support experimental evaluation and comparative analysis in later stages of the project.

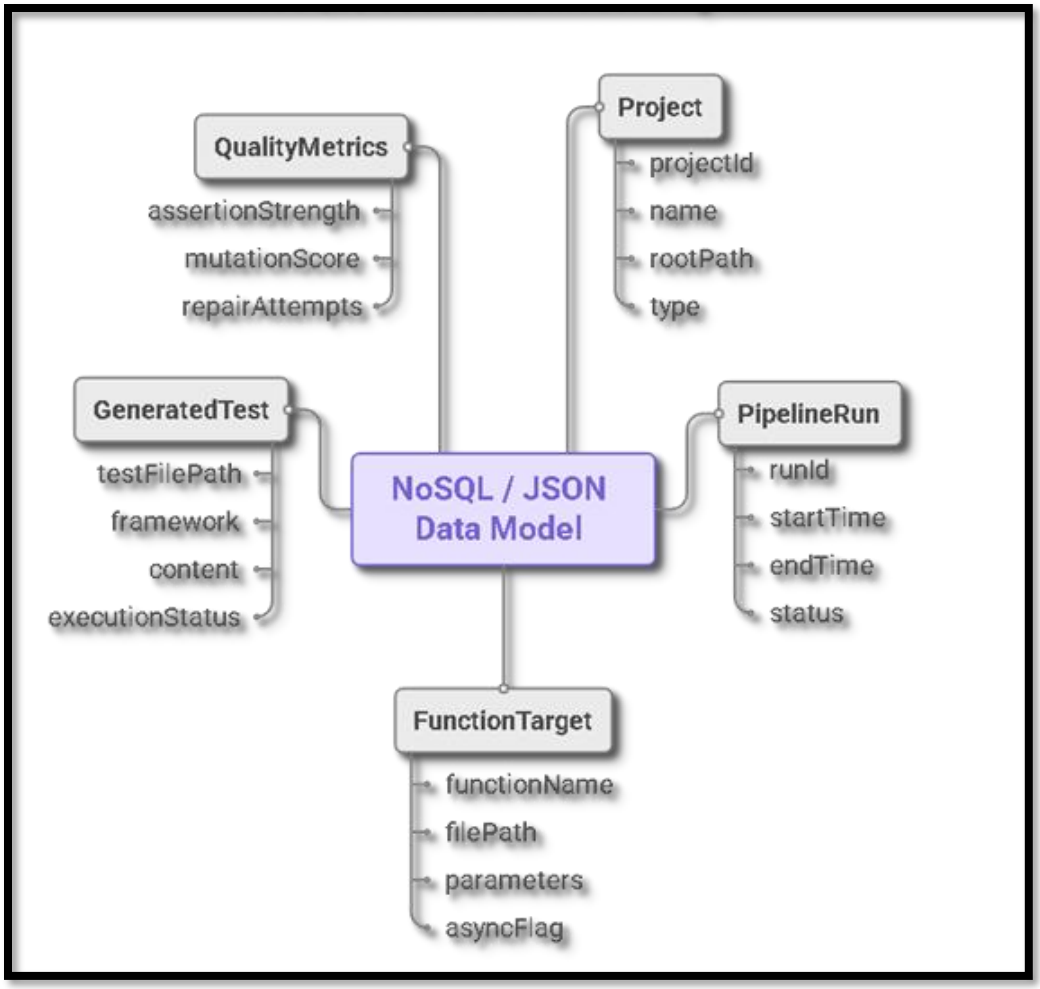

Figure 4-2 Conceptual NoSQL data model

4.3 Workflow Diagram

The end-to-end execution workflow of the system is shown in Figure 4-2.

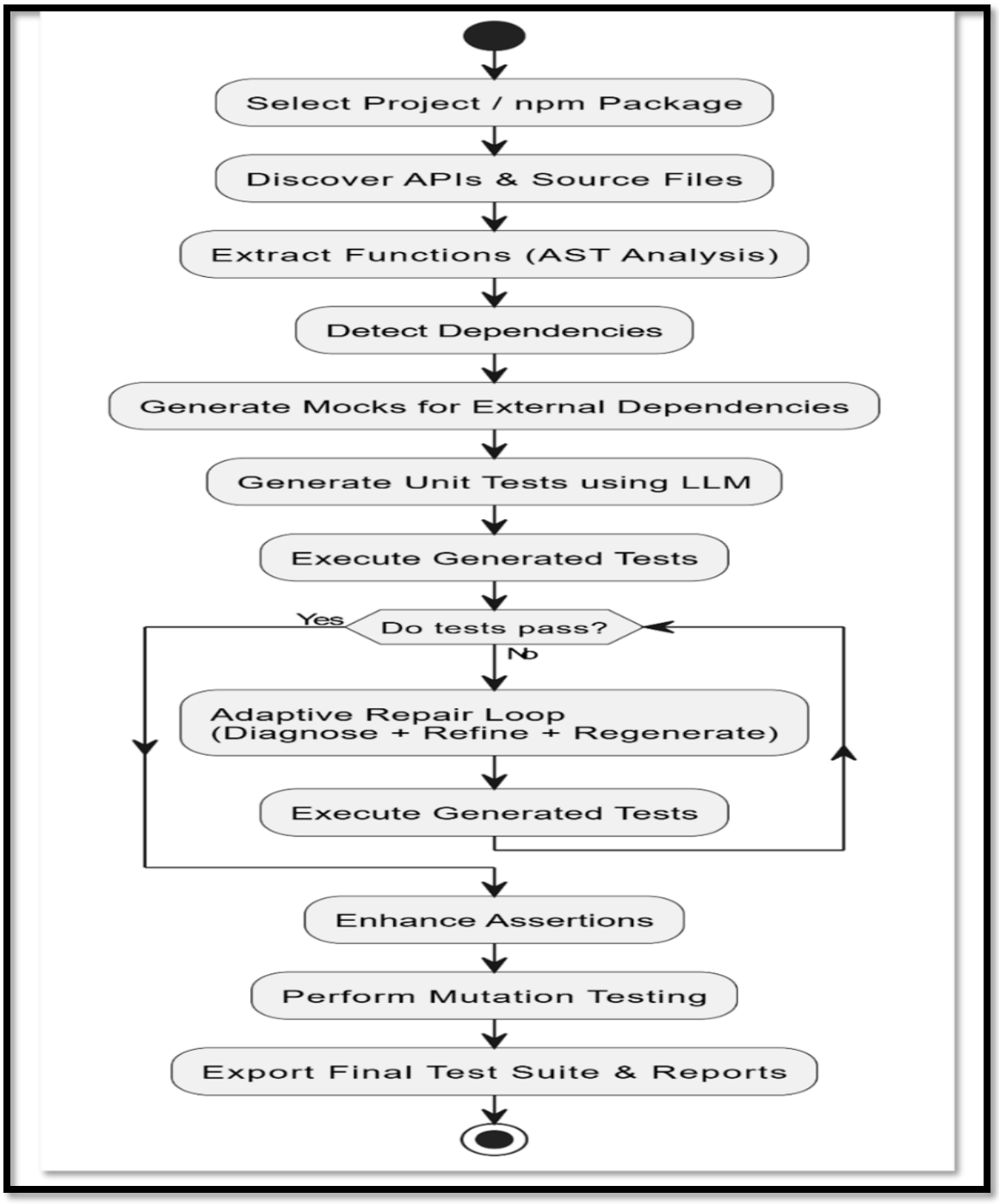

Figure 4-3 UML Activity Diagram

The workflow begins with selecting a project or npm package, followed by source file discovery and AST-based function extraction. The system then detects external dependencies and generates mocks automatically.

Next, unit tests are generated using a Large Language Model and executed. If the tests fail, the system enters an adaptive repair loop, where failures are diagnosed and tests are regenerated and re-executed. Once tests pass, assertion enhancement and mutation testing are performed to improve test quality.

Finally, the system exports the finalized test suite along with quality reports. The workflow explicitly models iteration and decision points, reflecting the verification-driven nature of the proposed solution.

4.4 Third-Parties Depedencies

The system integrates several third-party tools and libraries to support analysis, test generation, execution, and verification:

- **Node.js**: Runtime environment for system execution.
- **Babel Parser & Traverse**: Used for AST-based static code analysis.
- **Large Language Model APIs (e.g., Ollama / GPT-based models)**: Used for test logic generation and repair.
- **Testing Frameworks (Jest / Mocha)**: Used for executing generated unit tests.
- **Mocking Libraries (Sinon / Nock / Jest Mocks)**: Used for isolating external dependencies.
- **Mutation Testing Tools (StrykerJS)**: Used to evaluate fault-detection capability.
- **VS Code Extension APIs**: Planned for developer interface integration.

These dependencies enable the system to remain modular, extensible, and compatible with real-world JavaScript development environments.

Chapter 5

5. Software Development

This chapter describes the implementation details of the proposed unified framework for LLM-based automated unit test generation. While Chapter 3 and Chapter 4 focused on requirements and system design, this chapter explains how those concepts were realized in code, down to the module and function level.

The implementation reflects the architectural design presented in Chapter 4, ensuring a clear correspondence between design and implementation, which is critical for the correctness and evaluation of the Final Year Project. The chapter also discusses challenges encountered during development and how they were addressed.

At the current stage (40–50% completion), the implementation focuses on static analysis, dependency detection, mock generation, test template creation, and LLM-based test logic generation. Execution-based verification, adaptive repair, and assertion enhancement are planned for the next phase.

5.1 Coding Standards

To maintain readability, consistency, and maintainability, the following coding standards were followed throughout the project:

- **Indentation:**
  Four-space indentation was used consistently across all JavaScript files.
- **Naming Conventions:**
  - camelCase for variables and function names
  - PascalCase for conceptual entities and modules
  - File names reflect module responsibility (e.g., functionExtractor.js, dependencyDetector.js)
- **Modular Structure:**
  The codebase is organized into logically separated modules (parser, dependency, mock, LLM, test generation), following the single-responsibility principle.
- **Statement Standards:**

- ES6+ syntax is used consistently
- Arrow functions are preferred where appropriate
- Clear separation between pure functions and side-effect functions

These standards ensure that the system remains easy to understand, extend, and debug.

5.2 Development Environment

The project was developed using the following environment and tools:

Development Environment

- **Operating System:** Windows / Linux
- **Runtime:** Node.js (LTS version)
- **Package Manager:** npm
- **Editor:** Visual Studio Code

Technologies and Libraries

- **@babel/parser & @babel/traverse**
  Used for parsing JavaScript source code into Abstract Syntax Trees (AST) and performing static analysis.

- **Large Language Model Interface (Ollama/ GPT 3.5 turbo)**
  Used to invoke a local LLM for generating unit test logic based on structured prompts.

- **Testing Frameworks (Jest / Mocha)**
  Jest is currently used for test template generation, while Mocha-based execution is planned for the verification phase.

- **Mocking Libraries (Jest Mocks / Sinon / Nock)**
  Used to isolate external dependencies in generated unit tests.

The development environment was deployed locally by installing Node.js and project dependencies via npm install. This setup ensures portability and reproducibility across systems.

5.3 Software Description

This section describes the **major modules** of the implemented system. The modules correspond directly to the architectural layers described in Chapter 4.

Major Modules

- Source Code Parser Module
- Dependency Detection Module
- Mock Generation Module
- LLM-Based Test Generation Module

Each module is discussed below using selected code snippets that are critical to system operation.

5.3.1 Snippet 1: Source Code Upload and Validation

(User Story: Upload Source Code - E1-US1)

Code Snippet

```
function validateSourceFolder(files) {
  const jsFiles = files.filter(f => f.endsWith(".js"));
  if (jsFiles.length === 0) {
    throw new Error("No JavaScript or Node.js source files found.");
  }
  return true;
}
```

Description

This function validates the uploaded project directory by checking for the presence of JavaScript source files. If no valid `.js` files are found, the system throws an error and rejects the input. This ensures that only valid Node.js source code is accepted for further analysis.

Test Case: Source Code Upload and Validation

Table 9 TC1: Source Code Upload and Validation

Field	Description
Test ID	E1-US1-TC1
US ID	E1-US1

Test Case Description	Verify that valid Node.js source code is accepted and invalid input is rejected
Inputs	Valid Node.js folder with `.js` files; invalid folder with no JavaScript files
Expected Result	Valid input accepted; invalid input rejected with error
Actual Result	Validation performed correctly
Status	Passed

5.3.2 Snippet 2: Detection of External Dependencies

(User Story: Detect External Dependencies – E3-US1)

Code Snippet

```
function detectDependencies(functionBody, importMap) {
  return Object.keys(importMap)
    .filter(id => functionBody.includes(id))
    .map(id => importMap[id]);
}
```

Description

This function detects external dependencies used inside a function by matching imported identifiers against the function body. The output is a list of external dependencies that the function relies on, such as file system or HTTP libraries.

Test Case: Detection of External Dependencies

Table 5-10 TC2: Detection of External Dependencies

Field	Description
Test ID	E3-US1-TC1
US ID	E3-US1
Test Case Description	Verify detection of external dependencies in project code
Inputs	Project using `fs.readFileSync` and `axios.get`
Expected Result	Dependencies detected and classified correctly
Actual Result	Dependencies detected successfully
Status	Passed

5.3.3 Snippet 3: Mock Template Generation

(User Story: Generate Mocks – E3-US2)

Code Snippet

```
function generateMockTemplate(dependency) {
  if (dependency.type === "filesystem") {
    return `sinon.stub(fs, 'readFileSync');`;
  }
  if (dependency.type === "http") {
    return `nock('${dependency.baseUrl}').get('/').reply(200);`;
  }
}
```

Description

This function generates framework-specific mock templates for detected external dependencies. File system calls are mocked using Sinon, while HTTP calls are mocked using Nock. This ensures that generated unit tests remain isolated from real external resources.

Test Case: Mock Template Generation

Table 11 TC3: Mock Template Generation

Field	Description
Test ID	E3-US2-TC1
US ID	E3-US2
Test Case Description	Verify that mock templates are generated for dependencies
Inputs	Dependency list from TC2
Expected Result	Sinon stub for `fs.readFileSync`, Nock mock for `axios.get`
Actual Result	Mock templates generated correctly
Status	Passed

5.3.4 Snippet 4: LLM-Based Unit Test Generation

(User Story: Generate Unit Tests – E2-US1)

Code Snippet

```
async function generateTestWithLLM(prompt) {
  const response = await callLLM(prompt);
  return sanitizeOutput(response);
}
```

Description

This function sends a structured prompt containing function logic, dependency mocks, and test templates to the LLM. The LLM generates test logic, which is then sanitized and injected into the test template to produce an executable unit test.

Test Case: LLM-Based Unit Test Generation

Table 12-4 TC4: LLM-Based Unit Test Generation

Field	Description
Test ID	E2-US1-TC1
US ID	E2-US1
Test Case Description	Verify LLM-based unit test generation
Inputs	JavaScript function with simple logic
Expected Result	At least one valid unit test generated
Actual Result	Test generated successfully
Status	Passed

5.4 Implementation Challenges and Solutions

Several challenges were encountered during implementation:

- **Complexity of AST Traversal:**

Handling different function syntaxes required careful traversal logic. This was addressed using Babel's traversal utilities.

- **Dependency Resolution:**
  Identifying external dependencies accurately required combining import mapping with function-level usage analysis.
- **LLM Output Variability:**
  LLM-generated code was sometimes incomplete or inconsistent. This was mitigated by enforcing structured prompts and post-processing generated output.
- **Scope Management:**
  To avoid overambitious goals, execution-based verification and mutation testing were deferred to the next development phase.

A significant portion of development time was spent addressing these challenges, which directly contributed to the robustness of the current implementation.

Chapter 6

6. Software Deployment

This chapter describes how the developed system is prepared, configured, and deployed for execution in its intended environment. Since the proposed framework is a developer-oriented tool for automated unit test generation, the deployment strategy focuses on local execution in a controlled development environment. This approach ensures ease of use, reproducibility, and compatibility with real-world JavaScript and Node.js projects.

The chapter outlines the environment requirements, installation steps, and configuration details necessary to run the system successfully. Additionally, potential deployment challenges and guidelines for future scalability and maintenance are discussed.

6.1 Installation / Deployment Process Description

The proposed system is deployed as a **local Node.js-based application**. The deployment process involves setting up the runtime environment, installing required dependencies, and configuring the Large Language Model (LLM) interface. The following steps describe the installation and deployment procedure.

Step 1: Environment Requirements

Before installation, the following software and hardware requirements must be met:

- **Operating System:** Windows, Linux, or macOS
- **Runtime Environment:** Node.js (LTS version)
- **Package Manager:** npm
- **Memory:** Minimum 8 GB RAM (recommended for LLM interaction)
- **Disk Space:** At least 2 GB free space

These requirements ensure stable execution of static analysis, test generation, and LLM interaction.

Step 2: Install Node.js and npm

Node.js must be installed to execute the system.

1. Download Node.js (LTS version) from the official website.
2. Verify installation using the following commands:

```
3. node -v
4. npm -v
```

Successful execution of these commands confirms that Node.js and npm are correctly installed.

Step 3: Project Setup

1. Extract the project source code into a local directory.
2. Open a terminal or command prompt in the project root directory.
3. Install all required dependencies using:

```
4. npm install
```

This command installs all third-party libraries specified in the `package.json` file, including AST parsers, utility libraries, and test-related tools.

Step 4: LLM Configuration (Ollama Setup)

The system uses a **locally hosted LLM** through Ollama for generating unit test logic.

1. Install Ollama from the official website.
2. Start the Ollama service.
3. Pull the required model using:

```
4. ollama pull llama2
```

5. Ensure the Ollama API is running on:

```
6. http://localhost:11434
```

The system communicates with the LLM through this local API endpoint.

Step 5: Running the System

After completing the setup:

1. Place the target Node.js project or source file in the appropriate input directory.
2. Execute the system using:

```
3. npm start
```

4. The system performs:
    - Source code validation
    - Function extraction
    - Dependency detection
    - Mock generation
    - Test template generation
    - LLM-based test logic generation

Generated unit tests are saved automatically in the designated output directory.

Step 6: Verifying Output

After execution:

- Generated test files can be reviewed in the output folder.
- Logs confirm successful completion of each pipeline stage.
- Any errors during validation or generation are reported with clear messages.

This confirms that the system has been deployed and executed successfully.

6.2 Deployment Challenges and Limitations

Several challenges were encountered during deployment:

- **LLM Resource Usage:**
  Local LLM inference requires sufficient memory and CPU resources, which may limit performance on low-end systems.
- **Environment Compatibility:**
  Differences in operating systems and Node.js versions required careful dependency management to ensure portability.
- **Dependency Availability:**
  Missing or outdated npm packages could cause installation failures, addressed by maintaining a clean dependency list.

Despite these challenges, the system was successfully deployed in a local development environment.

6.3 Scalability and Maintenance Considerations

The current deployment strategy is suitable for individual developers and academic evaluation. For future scalability, the system can be extended to:

- Support **cloud-based LLM APIs** for improved performance
- Integrate with **CI/CD pipelines**
- Provide a **VS Code extension** for seamless developer interaction
- Persist execution data using a document-based NoSQL store

Modular design and clear separation of concerns ensure that the system can be maintained.

References

[1] M. Schäfer, S. Nadi, A. Eghbali, F. Tip, “An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation”, IEEE Transactions on Software Engineering, vol. 50, no. 1, pp. 85–105, Jan. 2024.

[2] G. Petrović, D. Ivanković, G. Fraser, R. Just, “Practical Mutation Testing at Scale: A View from Google”, IEEE Transactions on Software Engineering, vol. 48, no. 10, pp. 3900–3912, Oct. 2022.

[3] A. B. Sánchez, P. Delgado-Pérez, I. Medina-Bulo, S. Segura, “Mutation Testing in the Wild: Findings from GitHub”, Empirical Software Engineering, vol. 27, no. 132, pp. 1–35, Jul. 2022.

[4] A. B. Sánchez, J. A. Parejo, S. Segura, A. Durán Toro, M. Papadakis, “Mutation Testing in Practice: Insights from Open-Source Software Developers”, IEEE Transactions on Software Engineering, vol. 50, no. 5, pp. 2337–2349, Jul. 2024.

[5] S. Alagarsamy, C. Tantithamthavorn, A. Aleti, “A3Test: Assertion-Augmented Automated Test Case Generation”, Information and Software Technology, vol. 176, Art. no. 107565, Aug. 2024.

[6] B. Mali, K. Maddala, V. Gupta, S. Reddy, C. Karfa, R. Karri, “ChIRAAG: ChatGPT-Informed Rapid and Automated Assertion Generation”, arXiv preprint arXiv:2402.00093, 2024.

[7] F. Tip, J. Bell, M. Schäfer, “LLMorpheus: Mutation Testing using Large Language Models”, IEEE Transactions on Software Engineering, vol. 51, no. 6, pp. 1645–1665, Jun. 2025.

[8] J. Park, S. An, D. Youn, G. W. Kim, S. Ryu, “JEST: N+1-Version Differential Testing of JavaScript Engines and the Specification”, in Proc. 43rd IEEE/ACM International Conference on Software Engineering (ICSE 2021), 2021, pp. 156–157.

[9] M. Olsthoorn, D. Stallenberg, A. Panichella, “SynTest-JavaScript: Automated Unit-Level Test Case Generation for JavaScript”, in Proc. ACM/IEEE Intl. Workshop on Search-Based and Fuzz Testing (SBFT 2024), Apr. 2024, pp. 1–4.

[10] D. Stallenberg, M. Olsthoorn, A. Panichella, “Guess What: Test Case Generation for JavaScript with Unsupervised Probabilistic Type Inference”, in Proc. 14th Intl. Symposium

on Search-Based Software Engineering (SSBSE 2022), Lecture Notes in Computer Science, vol. 13711, pp. 67–82, Nov. 2022.

[11] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, J. Yin, “ChatUniTest: A Framework for LLM-Based Test Generation”, in Proc. 32nd ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering (FSE 2024), Demo Track, Jul. 2024.

[12] L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang, J. Chen, “On the Evaluation of Large Language Models in Unit Test Generation”, in Proc. 39th IEEE/ACM Intl. Conference on Automated Software Engineering (ASE 2024), Nov. 2024.

[13] S. Primbs, B. Fein, G. Fraser, “AsserT5: Test Assertion Generation Using a Fine-Tuned Code Language Model”, in Proc. 17th IEEE Intl. Workshop on Automating Software Test (AST 2025), 2025, pp. 12–23.

[14] Z. Nan, Z. Guo, K. Liu, X. Xia, “Test Intention Guided LLM-Based Unit Test Generation”, in Proc. 47th IEEE/ACM International Conference on Software Engineering (ICSE 2025), 2025.

[15] Y. Wang, C. Xia, W. Zhao, J. Du, C. Miao, Z. Deng, P. S. Yu, C. Xing, “ProjectTest: A Project-level LLM Unit Test Generation Benchmark and Impact of Error Fixing Mechanisms”, arXiv:2502.06556, 2025.

[16] Myers, G.J., Sandler, C., & Badgett, T. (2011). The Art of Software Testing. Wiley.

[17] Pressman, R.S., & Maxim, B.R. (2020). Software Engineering: A Practitioner’s Approach. McGraw-Hill.

[18] Rafiq, M., et al. (2021). “Automation of Acceptance and Integration Testing in Agile Environments.” IEEE Access, 9, 115732–115745.

[19] OpenAI, “GPT-3.5 Turbo Model,” OpenAI Platform Documentation, 2024. Available: https://platform.openai.com/docs/models/gpt-3-5. Accessed: Oct. 3, 2025.

[20] Y. Wang *et al.*, “CodeT5+: Open Code Large Language Models for Code Understanding and Generation,” *arXiv preprint,* arXiv:2305.07922, 2023.

[21] L. Allal *et al.*, “StarCoder: May the Source Be with You!,” *arXiv preprint,* arXiv:2305.06161, 2023.

[22] Meta AI, “Introducing Code Llama, a state-of-the-art large language model for coding,” Meta AI Blog, August 24, 2023. Available: https://ai.meta.com/blog/code-llama-large-language-model-coding/. Accessed: Oct. 3, 2025.

[22] Meta AI, “Introducing Code Llama, a state-of-the-art large language model for coding,” Meta AI Blog, Aug. 24, 2023. [Online]. Available: https://ai.meta.com/blog/code-llama-large-language-model-coding/. Accessed: Oct. 3, 2025.

[23] S. A. Abdallah, R. Moawad, and E. E. Fawzy, “An Optimization Approach for Automated Unit Test Generation Tools Using Multi-Objective Evolutionary Algorithms,” *Forensic Computer Information Journal*, vol. 2, no. 4, pp. 1–10, 2018.

[24] P. Abrahamsson, P. Kruchten, P. Gregory, et al., “ChatGPT as a Fullstack Web Developer – Early Results,” in P. Kruchten and P. Gregory, Eds., *XP 2022/2023 Workshops*, Springer, 2024, pp. 201–209.

[25] ACM, “ACM Digital Library,” 2024. [Online]. Available: https://dl.acm.org/

[26] B. K. Aichernig and K. Havelund, “AI-Assisted Programming with Test-Based Refinement,” in B. Steffen, Ed., *AISoLA 2023*, Springer, 2025, pp. 385–411.

[27] S. Banerjee and K. Patel, “Automated Unit Test Generation via Chain-of-Thought Prompt and Reinforcement Learning from Coverage Feedback,” in *Proc. ICSE 2025*, 2025.

[28] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The Oracle Problem in Software Testing: A Survey,” in *Proc. International Conference on Software Engineering (ICSE)*, 2014, pp. 119–122.

[29] S. M. B. Bhargavi and V. Suma, “A Survey of the Software Test Methods and Identification of Critical Success Factors for Automation,” *SN Computer Science*, vol. 3, no. 449, 2022.

[30] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, “Effective Test Generation Using Pre-Trained Large Language Models and Mutation Testing,” *Information and Software Technology*, vol. 170, Art. no. 107468, 2024.

[31] L. Da Silva, P. Borba, T. Maciel, W. Mahmood, T. Berger, J. Moisakis, A. Gomes, and V. Leite, “Detecting Semantic Conflicts with Unit Tests,” *Journal of Systems and Software*, vol. 212, Art. no. 112070, 2024.

[32] Elsevier, “ScienceDirect,” 2024. [Online]. Available: https://www.sciencedirect.com/

[33] K. Etemadi, M. Sirjani, M. H. Moghadam, P. Strandberg, and P. Pettersson, “LLM-Based Property-Based Test Generation for Guardrailing Cyber-Physical Systems,” in B. Steffen, Ed., *AISoLA 2025*, Springer, 2026, pp. 18–46.

[34] V. Garousi, S. Bauer, and M. Felderer, “NLP-Assisted Software Testing: A Systematic Mapping of the Literature,” *Information and Software Technology*, vol. 127, Art. no. 106321, 2020.

[35] Google, “JaCoCo: Java Code Coverage Library,” 2024. [Online]. Available: https://www.jacoco.org/

[36] S. Gu, C. Fang, Q. Zhang, F. Tian, and Z. Chen, “TestART: Improving LLM-Based Unit Test via Co-Evolution of Automated Generation and Repair Iteration,” *ACM Conference Proceedings*, 2024.

[37] Z. Guo, S. Chen, X. Xu, and X. Chen, “PC-TRT: A Test Case Reuse and Generation Tool to Achieve High Path Coverage for Unit Test,” *SoftwareX*, vol. 26, Art. no. 101918, 2024.

[38] I. Hajri, A. Goknil, F. Pastore, and L. C. Briand, “Automating System Test Case Classification and Prioritization for Use Case-Driven Testing in Product Lines,” *Empirical Software Engineering*, 2020.

[39] M. Harman et al., “Mutation-Guided LLM-Based Test Generation at Meta,” in *Proc. FSE Companion 2025*, 2025, pp. 180–191.

[40] IEEE, “IEEE Xplore Digital Library,” 2024. [Online]. Available: https://ieeexplore.ieee.org/

[41] M. Kaur, R. Bhatia, and V. Kumar, “Object-Oriented Test Case Generation Using Teaching-Learning-Based Optimization Algorithm,” *IEEE Access*, vol. 10, pp. 111345–111358, 2022.

[42] S. M. Khandaker, F. Kifetew, D. Prandi, and A. Susi, “AugmenTest: Enhancing Tests with LLM-Driven Oracles,” in *Proc. ICST 2025*, 2025.

[43] D. Kim, H. Park, and S. Ryu, “Nessie: Automatically Testing JavaScript APIs with Asynchronous Callbacks,” in *Proc. ICSE 2022*, 2022, pp. 1204–1215.

[44] J. Li, M. Zhou, and K. Hu, “An Empirical Investigation on the Readability of Manual and Generated Test Cases,” in *Proc. ICSE 2018*, 2018, pp. 390–401.

[45] Q. Li, Y. Wang, and Z. Xu, “Software Testing with Large Language Models: Survey, Landscape, and Vision,” *IEEE Transactions on Software Engineering*, vol. 51, no. 5, pp. 1405–1427, 2024.

[46] H. Liu, J. Zhang, and M. Zhao, “A System for Automated Unit Test Generation Using Large Language Models,” in *Proc. ICSTW 2025*, 2025.

[47] X. Luo, Y. Tang, Z. Liu, and Z. Zhou, “ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation,” *IEEE Transactions on Software Engineering*, vol. 50, no. 7, pp. 1302–1314, 2024.

[48] N. Mani and S. Attaranasl, “Adaptive Test Healing Using LLM/GPT and Reinforcement Learning,” in *Proc. ICSTW 2025*, 2025.

[49] M. Mock, J. Melegati, and B. Russo, “Generative AI for Test-Driven Development: Preliminary Results,” in *XP 2024 Workshops*, 2025.

[50] MuJava, “Mutation Analysis Tool for Java,” 2024. [Online]. Available: https://cs.gmu.edu/~offutt/mujava/

[51] MutPy, “Mutation Testing Tool for Python,” 2024. [Online]. Available: https://github.com/mutpy/mutpy

[52] T. Nguyen, L. Zhao, and W. Chen, “Automated Assertion Generation via Information Retrieval and Integration with Deep Learning,” in *Proc. ICSE 2022*, 2022.

[53] D. Olianas, M. Leotta, and F. Ricca, “MATTER: A Tool for Generating End-to-End IoT Test Scripts,” *Software Quality Journal*, vol. 30, pp. 389–423, 2022.

[54] OpenClover, “OpenClover Code Coverage Tool,” 2024. [Online]. Available: https://openclover.org/

[55] C. Paduraru, A. Staicu, and A. Stefanescu, “LLM-Based Methods for the Creation of Unit Tests in Game Development,” *Procedia Computer Science*, vol. 246, pp. 2459–2468, 2024.

[56] A. Panichella et al., “Small-Amp: Test Amplification in a Dynamically Typed Language,” *Empirical Software Engineering*, 2016.

[57] J. C. Pancher, J. Melegati, and E. M. Guerra, “Exploratory Test-Driven Development Study with ChatGPT,” in *XP 2025*, 2025.

[58] D. Planötscher, “NLP and GenAI in Agile Project Management: A Systematic Mapping Study,” in *XP 2026 Workshops*, 2026, pp. 41–49.

[59] ScalaCheck, “Property-Based Testing Tool for Scala,” 2024. [Online]. Available: https://github.com/typelevel/scalacheck

[60] J. Smith and R. Lee, “Reimagining Unit Test Generation with AI: A Journey from Evolutionary Models to Transformers,” *IEEE Access*, vol. 13, pp. 54780–54792, 2025.

[61] Springer, “SpringerLink,” 2024. [Online]. Available: https://link.springer.com/

[62] J. Sun, Y. Qian, and P. Yang, “Exploring the Capability of ChatGPT in Test Generation,” in *Proc. QRS-C 2023*, 2023.

[63] C. Tan, A. Lim, and D. Wang, “StubCoder: Automated Generation and Repair of Stub Code for Mock Objects,” in *Proc. ESEC/FSE 2023*, 2023.

[64] M. Taromirad and P. Runeson, “Assertions in Software Testing: Survey, Landscape, and Trends,” *International Journal on Software Tools for Technology Transfer*, vol. 27, pp. 117–135, 2025.

[65] Z. Wang, K. Liu, and G. Li, “HITS: High-Coverage LLM-Based Unit Test Generation via Method Slicing,” in *Proc. FSE 2025*, 2025.