LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

DEPARTMENT OF COMPUTER SCIENCE -
SOFTWARE AND COMPUTATIONAL SYSTEMS LAB

Towards Automated Software Testing - Applying TextGrad to Test Suite Generation using LLMs

Master Thesis

Moritz Gärtner

Supervisor: Prof. Dr. Dr. h.c. Martin Wirsing

Advisor: Dr. Lenz Belzner

Submission Date: November 10th, 2025

EIGENSTÄNDIGKEITSERKLÄRUNG

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich aus einer Literatur oder anderen Quellen übernommen habe, sind eindeutig als Zitate mit Quellenangabe gekennzeichnet.
Ich habe ChatGPT-5 ausschließlich zur Verbesserung der Lesbarkeit und Verständlichkeit des vorliegenden Textes in Einklang mit wissenschaftlichen Standards verwendet.
Diese gedruckte Fassung ist ein Ausdruck der eingereichten elektronischen Fassung.

München, 10. November 2025

Ort, Datum | Unterschrift

DISCLAIMER

I hereby declare that I have written this work independently and only with the aids specified. All passages that I have taken from literature or other sources have been clearly marked as quotations with reference to the source.
I used ChatGPT-5 exclusively for language refinement, such as improving readability and rephrasing for clarity in an academic writing style in accordance with standard academic practices for language polishing.
This printed copy is a printout of the submitted electronic copy.

Munich, November 10th, 2025

Location, Date | Signature

DANKSAGUNGEN

Mein besonderer Dank gilt Herrn Prof. Dr. Dr. h.c. Martin Wirsing sowie Dr. Lenz Belzner für ihre engagierte Betreuung, ihre stetige Unterstützung und die vielen hilfreichen Anregungen im Verlauf dieser Arbeit. Ihre fachliche Expertise und ihre Bereitschaft, Zeit für konstruktives Feedback zu investieren, haben maßgeblich zur Qualität dieser Arbeit beigetragen.

Ebenso möchte ich mich bei meinen Kolleginnen, Kollegen und Freunden bedanken, die mich während der Entstehung dieser Arbeit begleitet haben. Durch ihre Rückmeldungen, Diskussionen und den gemeinsamen Austausch konnte ich zentrale Aspekte weiter schärfen.

Mein größter Dank gilt schließlich meinen Eltern und meiner Familie. Sie haben mich während meiner gesamten akademischen Laufbahn begleitet, mich motiviert und mir Rückhalt bei allen Herausforderungn gegeben. Ihre bedingungslose Unterstützung und ihr Vertrauen in meinen Weg haben diese Arbeit erst möglich gemacht.

ACKNOWLEDGMENTS

I would like to express my deep gratitude to Prof. Dr. Dr. h.c. Martin Wirsing and Dr. Lenz Belzner for their dedicated guidance, continuous support, and the many insightful suggestions offered throughout the course of this work. Their expertise and their readiness to provide thoughtful, constructive feedback played a crucial role in shaping the quality of this thesis.

I am also grateful to my colleagues and friends who accompanied me during this process. Their feedback, discussions, and shared reflections helped me to refine key ideas and strengthen the overall direction of the work.

Above all, I extend my deepest thanks to my parents and my family. Their encouragement, patience, and unwavering belief in my path have supported me throughout my academic journey. Without their trust and constant support, this thesis would not have been possible.

ABSTRACT

The rapid progress in Artificial Intelligence (AI) and particularly Large Language Models (LLMs) has fundamentally reshaped the field of Software Engineering (SE). Recent advances demonstrate that LLMs are not only capable of generating isolated code snippets but can perform coherent reasoning across software projects, including testing, validation, and maintenance. This thesis explores the potential of LLMs to autonomously generate, execute, and iteratively refine comprehensive test suites for real-world software systems.

Building upon established SE principles and recent developments in LLM-based optimization frameworks such as *TextGrad* [39] and *TestART* [15], the thesis introduces **LIFT** (LLM-based Iterative Feedback-driven Test suite generation) - a novel, automated, and feedback-driven approach to test suite generation. Within LIFT, specialized LLM agents act as generator, debugger, and evaluator components that cooperatively evolve test suites through textual gradients, self-assessment, and refinement loops.

A case study on the Python library `simplejson` evaluates LIFT over multiple trials and iterations, analysing the evolution of test count, coverage, and mutation score. Results indicate that LLMs can consistently construct executable test suites, extend them meaningfully across iterations, and approach near-complete behavioral coverage when given sufficient computation power and context. These findings suggest that LLMs are capable of performing complex reasoning about software structure and behavior, moving beyond traditional unit test generation toward a holistic understanding of program correctness. Yet, to date, assertion quality is not on par, highlighting the continued impact of the Oracle Problem within LLM-based test generation.
The implemented framework is made available under: `Sarius32/LIFT`

The thesis contributes an empirical foundation for autonomous, LLM-driven quality assurance and discusses implications for scalability, tool integration, and future research. By aligning generative AI with core SE practices, it highlights a promising path toward self-improving software testing systems that bridge human expertise and machine reasoning.

Keywords: Large Language Models, Software Engineering, Software Testing, Test Case Generation, Test Suite Generation, TextGrad

ZUSAMMENFASSUNG

Der rasante Fortschritt im Bereich der Künstlichen Intelligenz (KI) und insbesondere der Large Language Models (LLMs) hat das Gebiet des Software Engineering (SE) grundlegend verändert. Jüngste Entwicklungen zeigen, dass LLMs nicht nur in der Lage sind, isolierte Codefragmente zu generieren, sondern auch kohärente Schlussfolgerungen über gesamte Softwareprojekte hinweg zu ziehen - einschließlich Testen, Validierung und anschließender Wartung. Diese Arbeit untersucht das Potenzial von LLMs, umfassende Testsuiten für Softwaresysteme autonom zu erzeugen, auszuführen und iterativ zu verbessern.

Aufbauend auf etablierten Prinzipien des Software Engineering und aktuellen Entwicklungen in LLM-basierten Optimierungsframeworks wie *TextGrad* [39] und *TestART* [15] stellt die Arbeit **LIFT** (LLM-based Iterative Feedback-driven Test suite generation) vor - einen neuartigen, automatisierten und auf Feedback aufbauenden Ansatz zur Generierung von Testsuiten. Innerhalb von LIFT handeln spezialisierte LLM-Agenten als Generator-, Debugger- und Evaluationskomponenten, die gemeinsam Testsuiten mithilfe von textuellen Gradienten, Selbstbewertung und wiederholter Verfeinerung weiterentwickeln.

Eine Fallstudie zum Python-Paket `simplejson` evaluiert LIFT über mehrere getrennte Durchläufe und Iterationen und analysiert die Entwicklung von Testanzahl, Testabdeckung und Mutationserkennung. Die Ergebnisse zeigen, dass LLMs in der Lage sind, konsistent ausführbare Testsuiten zu konstruieren, diese über Iterationen hinweg sinnvoll zu erweitern und bei ausreichender Rechenleistung nahezu vollständige Verhaltensabdeckung zu erreichen. Diese Erkenntnisse deuten darauf hin, dass LLMs komplexe Schlussfolgerungen über Softwarestrukturen und -verhalten ziehen können und damit über die traditionelle Unit-Test-Generierung hinaus zu einem ganzheitlichen Verständnis von Programmkorrektheit beitragen. Jedoch ist die Qualität der Assertions aktuell nicht hinreichend, was auf den anhaltenden Einfluss des Oracle Problems hindeutet.
Das implementierte Framework ist zu finden unter: `Sarius32/LIFT`

Die Arbeit liefert eine empirische Grundlage für autonome, LLM-gestützte Qualitätssicherung und diskutiert Implikationen hinsichtlich der Skalierbarkeit, von Tool-Integration und zukünftiger Forschung. Durch die Verbindung generativer KI mit zentralen Praktiken des Software Engineerings wird ein vielversprechender Weg zu selbstverbessernden Testsystemen aufgezeigt, die menschliche Expertise und maschinelles Schlussfolgern miteinander vereinen.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

AI Artificial Intelligence. 1–4, 16

API Application Programming Interface. 50

BDD Behavior-Driven Development. 10

CD Continuous Delivery. 9

CI Continuous Integration. 9

CLI Command-Line Interface. 49, 60

FLOP Floating-Point Operation. 2

FSS First Sufficient Test Suite. 50, 51, 54–64, 67, XVI–XVIII, XX, XXII

GUI Graphical User Interface. 16, 18, 52

JSON JavaScript Object Notation. 42–45, 59, 61

LIFT LLM-based Iterative Feedback-driven Test suite generation. 1, 4, 5, 30–33, 35, 38–40, 42, 46, 49, 50, 52–57, 59, 61–63, 65–68, I, IX

LLM Large Language Model. 1–5, 16–34, 42, 43, 45–48, 50–54, 56, 60–63, 65–68

LLM4SE Large Language Models for Software Engineering. 16, 17

LPS Last Passing Test Suite. 50, 51, 54–57, 59–63, 65, 67, XVI, XVII, XIX, XXI, XXIII

MuTAP Mutation Test case generation using Augmented Prompt. 22–25, 29, 30, 47, 62

PUT Program under Test. 1, 4, 10, 11, 13, 20, 22, 24, 25, 29–31, 35–37, 42, 46, 48–50, 52, 56, 60, 61, 65–68

QA Quality Assurance. 10, 14, 16

RE Requirements Engineering. 6, 16

RFC Requests for Comments. 43

SDLC Software Development Life Cycle. 6–8, 10, 11, 16

SE Software Engineering. 1, 4–6, 8, 10, 16–18, 29, 64, 65, 67, 68

STLC Software Testing Life Cycle. 6, 13–15, 19

TDD Test-Driven Development. 9, 10

TSD Test Suite Debugger. 32, 35–38, 52, 53, IV

TSE Test Suite Evaluator. 31–33, 35, 38–41, 50, 53–55, 57, 62, 64, VII

TSG Test Suite Generator. 31, 32, 35–38, 40, 50, 55, 60, 64, I

V&V Verification & Validation. 7, 30, 39

XP Extreme Programming. 9

1 Introduction

The rapid advances in Artificial Intelligence (AI) over the past years have fundamentally transformed the landscape of Software Engineering (SE). Once limited to automated assistance in isolated development tasks, intelligent systems have now become capable of performing end-to-end reasoning and code generation across entire projects [3]. This paradigm shift opens up new opportunities to rethink long-established engineering activities such as software testing, verification, and maintenance.

Building upon these developments, this thesis investigates how Large Language Models (LLMs) can be employed to autonomously construct, execute, and iteratively refine complete test suites for real-world software projects. By connecting modern LLM-based reasoning techniques with established Software Engineering principles, it explores a new level of automation that extends beyond single test generation toward comprehensive, self-improving test suites. The work thus positions itself at the intersection of software quality assurance, generative AI, and continuous improvement in automated testing.

This thesis begins with an overview of the foundations of Software Engineering and Software Testing in Section 2, discussing established verification and validation activities, test levels, and traditional testing methodologies that form the conceptual baseline for this work. Following this, Section 3 reviews recent developments in the use of LLMs for Software Engineering tasks, with particular attention to automated test generation and the growing evidence of models' ability to reason about program semantics and correctness. These studies serve to highlight both the promise and current limitations of existing approaches, motivating the need for a more systematic, feedback-driven method of test suite construction.

Building on this foundation, Section 4 introduces a novel framework named *LIFT* (LLM-based Iterative Feedback-driven Test suite generation), which extends the ideas of differentiable textual optimization and self-evaluative reasoning as previously explored in Section 3. LIFT enables models to iteratively analyze, refine, and expand their own test suites through structured feedback loops, thereby approaching software testing as an evolving optimization process rather than a static generation task. The framework is then applied to the open-source Python library `simplejson` - described in Section 5 - as the Program under Test (PUT), providing a realistic yet controlled environment to assess the models' ability to generate comprehensive and executable test suites.

The results are discussed in Section 6, which focuses on both quantitative and qualitative aspects of test generation. Metrics such as test count, coverage, and mutation score are examined and compared against contemporary approaches in the literature. The section also includes an analysis of threats to validity. Finally, Section 7 outlines potential extensions of this research, suggesting a broader range of tested projects, programming languages, and model families, as well as refined prompt-engineering strategies to enhance scalability and robustness. The thesis concludes in Section 8 by summarizing its contributions and highlighting how LIFT advances the current state of LLM-based Software Testing.

1.1 Motivation

Software has become an indispensable component of virtually every domain, yet its long-term maintainability continues to be a major concern. Over time, legacy systems accumulate technical debt, dependencies become outdated, and project-specific knowledge fades - leading to an increasing loss of maintainability and understandability [27, 34]. This *knowledge erosion* poses a threat to the sustainability of large software systems, especially when documentation or test coverage is incomplete.

At the same time, the capabilities of Artificial Intelligence have experienced exponential growth. Figure 1 illustrates the rapid increase in training compute[1] for large models over the last decade, reflecting both the scale and sophistication of modern LLMs. With the widespread adoption of systems like GitHub Copilot and ChatGPT, software development has entered a new era of AI-augmented productivity. These systems demonstrate that natural language models can already comprehend, generate, and reason about complex source code structures.

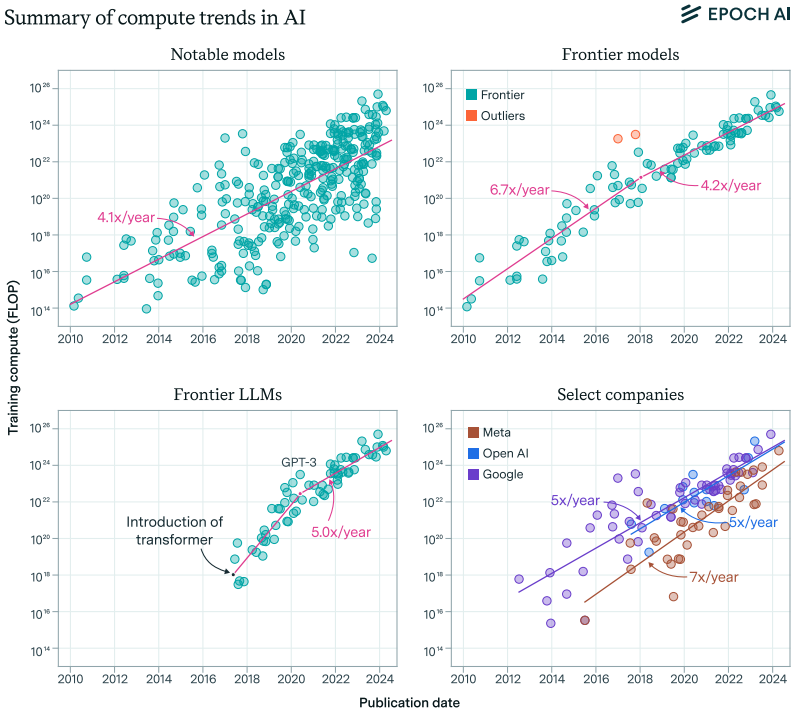


Figure 1: Summary of the compute growth trends for overall notable models, frontier models, top language models, and top models within leading companies [31]

The growing complexity of modern software systems demands testing strategies that can evolve dynamically alongside the code. If an artificial model can fully capture a program's behavior, two powerful possibilities emerge:

(a) automatic reimplementation or portation of software across languages or platforms, and
(b) verification through AI-driven behavioral testing that ensures functional equivalence.

[1]Training compute describes the amount of Floating-Point Operations (FLOPs) used during a model's training. [17]

This thesis focuses on the latter - leveraging LLMs to automatically generate, evaluate, and refine test suites as a means of verifying program identity and functionality.

In this context, Artificial Intelligence is not merely a supporting tool but a potential enabler for scalable software validation. By combining principles of software testing with the emerging LLM paradigm, this work aims to explore how LLM-based agents can autonomously drive test generation and refinement, bridging the gap between human insight and automated reasoning.

1.2 Research Questions

Building on the motivation outlined above, the following section defines the research questions that guide this thesis. The importance of software testing is well established and Large Language Models (LLMs) have recently emerged as powerful and widely adopted tools. This leads to the question of their adoptability in the realm of automated test generation. It is not yet clear whether LLMs are fundamentally capable of generating test suites, whether such test suites can be considered "good" in terms of established testing criteria, and how their performance compares to existing automated approaches. To address these uncertainties, the first research question focuses on the basic feasibility of employing LLMs for test generation:

RQ-1: Are LLMs able to generate test suites at all?

To determine whether Large Language Models are able to generate test suites at all, several fundamental aspects must be examined. A central requirement is the ability to provide or approximate suitable test oracles. Without meaningful assertions, test cases cannot validate program behavior, regardless of how well they are structured and implemented. It is therefore crucial to investigate whether LLMs can extrapolate expected outputs or conditions that go beyond trivial checks.

Equally important is the syntactic correctness of the generated code. Test cases must be valid and executable to be considered usable artifacts in the testing process. Code that fails to run cannot contribute to a functioning test suite. Even if the generated code is syntactically valid, it must also integrate correctly into its surrounding context. Handling imports, external dependencies, and framework-specific conventions such as `pytest` fixtures is necessary for tests to work within larger projects.

The manner in which these tests are produced also plays a central role. Some outputs may be usable immediately, while others may require refinement or fixes, either through human feedback or automated repair. Exploring this aspect helps to understand the autonomy and practicality of LLM-based test generation.

Finally, the scope of testing must be taken into consideration. Test cases target different levels of abstraction, from unit-level checks of individual functions to broader integration, system, or acceptance testing. Examining which of these levels LLMs are able to cover helps to determine the boundaries of their current capabilities and therefore their ability to generate full-scale test suites.

These considerations combined form the basis for answering RQ-1 and break it down into the following sub-questions:

RQ-1.1: Can LLMs provide or approximate suitable test oracles?

RQ-1.2: Can LLMs generate syntactically correct and executable test code?

RQ-1.3: Can LLMs understand the structure of the Program under Test, make correct references in the test code, and handle interactions with the working environment?

RQ-1.4: Can LLMs produce usable tests in a single shot or do they require iterative human/feedback guidance to generate runnable tests?

RQ-1.5: Which levels of testing can LLMs generate and where do their current capabilities reach their limits?

If the assumption holds true that RQ-1 can broadly be answered with "yes", this does not yet imply that the resulting test suites are of sufficient value. The mere ability to generate executable tests does not address their adequacy, effectiveness, or long-term usefulness. To assess these aspects, the second research question focuses on the quality of LLM-generated test suites.

RQ-2: Are LLMs able to generate "good" test suites?

A generated test suite that merely executes without compilation or runtime errors may provide little value if it fails to capture relevant behavior, is unreadable, or does not detect faults. Addressing the quality dimension, therefore, requires an investigation of the criteria by which test suites can be evaluated and how these criteria apply to LLM-generated artifacts.

A first step concerns the identification of suitable quantitative measures. While software testing research provides a variety of established metrics such as statement coverage, branch coverage, mutation score, or simply the number of generated tests, it is not immediately clear which of these are most informative overall. Some metrics may offer only limited insight into the effectiveness of a test suite, while others may capture more meaningful aspects of adequacy. Determining which measures are relevant and how they should be applied is, therefore, an important part of the investigation.

In addition to numerical indicators, qualitative properties also play a crucial role in determining the value of a test suite. Factors such as readability, maintainability, and similarity to human-written tests need to be examined in relation to their contribution to test quality. Equally important is whether the generated tests exhibit a true validation character. Are they meaningfully checking program behavior rather than merely executing code paths without substantive assertions? It remains to be analysed how such qualitative aspects can be evaluated and whether additional criteria beyond established practice need to be considered.

Building on these two perspectives, the question arises of what actually constitutes a "good" test suite in the specific context of LLM-based generation. It is not yet clear whether traditional notions of adequacy, oracle soundness, and maintainability apply directly or whether LLM-generated suites call for a redefinition of quality criteria. Clarifying this is necessary before meaningful evaluations can be carried out.

Once quantitative and qualitative criteria have been identified, the next step is to examine the extent to which LLM-generated test suites actually satisfy them. The investigation must therefore determine how well-established measures are fulfilled by LLM-generated suites. Additionally, any LLM-specific criteria need to be checked, and all metrics need to be weighted.

Finally, the ultimate purpose of any test suite is its ability to uncover faults. Coverage or readability alone does not guarantee effectiveness if real defects remain undetected. Investigating whether LLM-generated test suites can expose seeded defects or mutants, therefore provides the most substantial evidence of their practical value.

Taken together, these considerations form the basis for answering RQ-2 and are addressed through the following sub-questions:

RQ-2.1: Which quantitative metrics characterize good test suites?

RQ-2.2: Which qualitative criteria need to be evaluated for a good test suite?

RQ-2.3: What constitutes a good test suite in the context of LLM-based generation?

RQ-2.4: How well do LLM-generated test suites fulfill traditional and possible LLM-adapted quantitative and qualitative metrics?

RQ-2.5: To what extent are LLM-generated test suites able to detect faults beyond simply achieving high coverage?

2 Background

To understand the context in which this thesis is situated, this section introduces the foundational concepts of Software Engineering and the structure and dynamics of the Software Development Life Cycle. It situates Software Testing within Software Engineering and further outlines the Software Testing Life Cycle as a specialized process for ensuring quality and dependability. Additionally, it defines the role of Test Suites and their effectiveness with regards to Software Testing.

2.1 Software Engineering

"Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use." [34, p. 21]
Sommerville differentiates between the engineering aspect in Software Engineering - producing solutions to problems within time and financial limitations - and software production itself, which includes all other aspects apart from technical processes like project management or methods and tooling. Furthermore, he identifies four fundamental activities that are common to all software processes:

1. **Software specification**, in which engineers and customers define the required system functionality as well as constraints on its operation.
2. **Software development**, where the system is designed and implemented according to the specification.
3. **Software validation**, in which the system is checked and tested to ensure it satisfies customer requirements.
4. **Software evolution**, where the software is adapted and modified to meet changing customer needs and market conditions. [34, p. 23]

Requirements Engineering Requirements are a reflection of the stakeholders' needs and wishes for a system or software. They "establish a high-level view of what the system might do and the benefits that it might provide." [34, p. 104] Functional requirements describe capabilities and behavior. They detail technical designs and limitations and describe the type of software being developed. Non-functional requirements, in turn, "relate to emergent system properties such as reliability, response time, and memory use." [34, p. 107] Their aim is to provide a general idea of the system's attributes like performance, security, and usability, not implementation details. [5, 27, 34]

The process of creating and working with requirements is called Requirements Engineering (RE) and comprises a structured workflow that transforms stakeholder needs into controlled and testable specifications. The scope encompasses:

(1) elicitation: the systematic acquisition of goals, constraints, and domain knowledge from stakeholders and sources,

(2) analysis and negotiation: where conflicts are resolved, feasibility is assessed, and requirements are prioritized and allocated[2],

(3) specification: production of precise, unambiguous, and verifiable artifacts with formally defined requirement attributes (identifier, source, rationale, priority, verification method, acceptance criteria), and

(4) validation: checking the set for correctness, completeness, and consistency against stakeholder intent. [34, ch. 4]

On a different level, requirements are broken down into system and (high-/low-level) software requirements. In this, the system requirements provide general information on the whole system, while software requirements are derived from the system requirements and relate to individual software parts or components. [5, 34]

Verification & Validation (V&V) Verification & Validation constitutes a life cycle-spanning assurance discipline. Verification establishes conformance of artifacts to their specifications through analysis, inspection, demonstration, and test. Validation, on the other hand, establishes the fitness of the delivered system for its intended use in its operational context. NASA offers two easy questions: *"Did I build the product right?"* [24, p. 196] for verification and *"Am I building the right product?"* [24, p. 196] for validation.
V&V spans the entire life cycle and all artifacts: requirements, architectural and detailed designs, source code, tests, and accompanying documentation. Bidirectional traceability is central to Verification & Validation. To this end, a requirements traceability matrix can be used to link each requirement to its verification method, concrete test specifications and test cases, and finally to observed results. [5, 27, 34]

The effectiveness of V&V depends on its systematic integration with life cycle activities and artifacts, together with bidirectional traceability. To make these dependencies explicit, the following subsection introduces the Software Development Life Cycle as the coordinating structure that schedules V&V activities, defines their entry and exit criteria, and governs the flow of evidence across phases. [5, 27, 34]

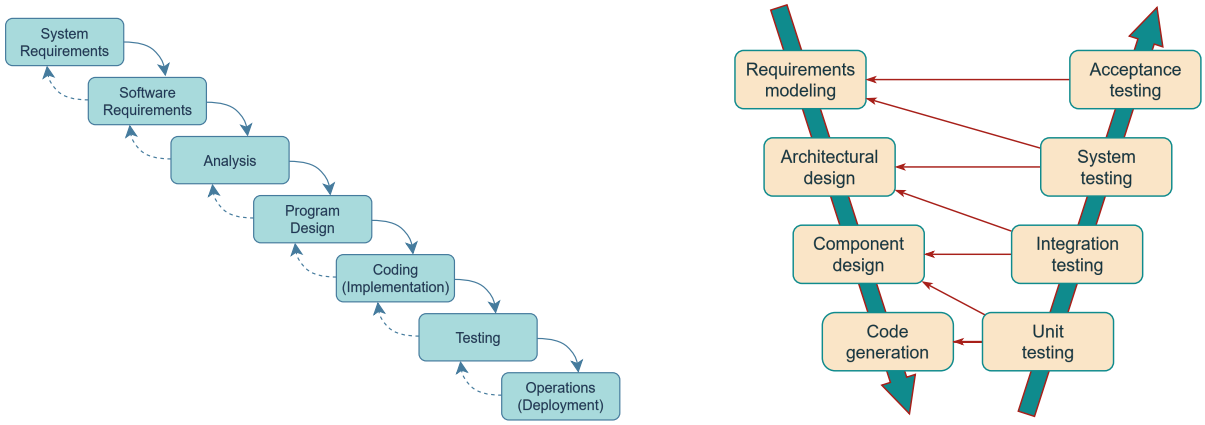
2.2 Software Development Life Cycle

Sommerville identifies the Software Development Life Cycle (SDLC) as "a simplified representation of a software process" [34, p. 45] that includes all the aforementioned fundamental activities. The SDLC provides a process architecture for software projects, typically spanning problem definition and requirements, architectural and detailed design, implementation, integration, verification, validation and acceptance, deployment, and operation with evolution or maintenance. Different development models instantiate this life cycle with distinct ordering, feedback, and documentation practices, trading off controllability, responsiveness to change, and early defect discovery. The role of Software Testing within these models highlights its importance and whether it is implemented in a static, iterative or incremental manner. [5, 34]

[2]Note that Sommerville does not include this step in [34, ch. 4]. Rather, Pressman mentions it in [27, p. 122].

Waterfall Model First described by Royce in 1970, the Waterfall Model depicts a linear flow from the early planning stages of a project, progressing through implementation to the deployment of software. Each phase must be completed before the next begins and feedback between phases is limited. Even though Royce originally did include so-called "feedback loops", Pressman highlights that most modern references to the Waterfall Model "treat it as if it were strictly linear" [27, p. 39]. Figure 2a shows all stages and possible loops as described by Royce. The model's strengths lie in its simplicity and clear structure, which make it suitable for projects with stable and well-understood requirements. However, its rigidity makes it less adaptable to changes during development and defects discovered late in the process can be costly to address. [1, 5, 27, 34]

Testing as the prior to last step within the Waterfall model highlights, that it is only considered late in the process of software development. Yet, it is necessary to provide feedback about defects found in the program possibly impacting the whole chain of previous activities. Projects developed based on this model therefore have a tendency to underestimate the effort of testing, based on the impression that, with well-implemented software, testing is only complementary. [5, 27, 34]



(a) Waterfall Model after Royce [28, p. 329/330]

(b) V-Model [27, p. 40]

Figure 2: Linear Software Development Life Cycle models

V-Model Extending the Waterfall Model, the V-Model explicitly links each development phase with a corresponding testing phase. Through this, two streams are created that form the shape of a "V", where the left side represents the decomposition of requirements and design into progressively more detailed specifications, while the right side illustrates the corresponding integration and testing activities. For instance, unit testing is directly associated with component design, integration testing with architectural design, and system and acceptance testing with the requirements analysis phase. Different V-Models exist, varying in the individual broken-down stages in the streams. The explicit mapping emphasizes early planning of verification and validation activities, fostering stronger traceability between requirements, design, and testing. The V-Model is particularly popular in safety-critical domains such as aerospace and automotive Software Engineering, where rigorous testing and documentation are mandatory. Pressman concludes that "there is no fundamental difference between the classic life cycle [Waterfall Model] and the V-Model" [27, p. 40]. Therefore,

both models assume stable requirements and are less flexible when confronted with frequent changes. Yet, the V-Model enforces the impression that testing needs to be thought of in every step of the software development, through the explicit linking between development and testing phases. [1, 4, 25, 27, 34]

Iterative/Incremental Models In contrast to the presented models, iterative models organize development as a sequence of repeated cycles delivering parts (increments) of software. This incremental delivery allows feedback to be incorporated early and continuously, reducing the risk of building a system that does not meet user needs. Iterative models are particularly well-suited for projects where requirements are uncertain or likely to evolve during development. Often, iterative models are equated with agile methods, which gained wide adoption after the publication of the Agile Manifesto [2] in 2001. Agile emphasizes flexibility, close collaboration between developers and customers, and rapid delivery of working software. Instead of completing all specifications and designs upfront, agile methods promote adaptive planning and continuous improvement. Agile has proven effective in dynamic environments where requirements frequently change and stakeholder involvement is high. However, agile methodologies can be challenging to apply in highly regulated or safety-critical domains, where comprehensive documentation, traceability, and formal verification are required. [1, 2, 25, 27, 34]

DevOps, Continuous Integration, and Continuous Delivery Since the 2010s, software development has increasingly adopted practices that emphasize automation, rapid delivery, and close collaboration between development and operations teams. The shift toward DevOps and continuous practices reflects a natural evolution from iterative and agile models, aiming to shorten release cycles while maintaining reliability and quality. DevOps integrates development and operations, breaking down traditional silos to enable faster delivery of software. Continuous Integration (CI) is an integral part of DevOps in which developers frequently merge code into a shared repository. Each integration triggers an automated build and test process, ensuring that issues are detected early and integration problems are minimized. Building on this shared work, Continuous Delivery (CD) extends automation into delivery pipelines, ensuring that every build is a potentially releasable product. For that, automatic testing and packaging are performed after passing all quality checks. Automated unit, integration, and system tests can be executed as part of the pipeline, reducing the cost of defects and ensuring consistent quality. Continuous monitoring in production environments further supports rapid detection of issues, feeding back into the development process to enable ongoing improvement. Through this, adaptability is improved and the time between concept and delivery is reduced. [1, 19, 27, 34]

Test-/Behavior-Driven Development Test-Driven Development (TDD) emerged as one of the core practices of Extreme Programming (XP) and is characterized by the principle of writing unit tests prior to implementing the corresponding functionality. "TDD develops the test cases as a surrogate for a software requirements specification document rather than as an independent [validation artifact]" [5, p. 94]. This practice guides the design of code by iteratively defining, implementing, and refining small units of functionality through automated test cases, thereby ensuring traceability between requirements, implementation, and verification. Beyond its verification role, TDD also

fosters clearer understanding and elaboration of user needs and software requirements through the continuous reformulation of test specifications. [5]

Behavior-Driven Development (BDD) extends and refines the principles of TDD by shifting the focus from testing implementation details towards specifying and validating the intended behavior of a system. According to Farooq et al., BDD emphasizes collaboration between technical and non-technical stakeholders by expressing requirements in structured, executable natural language scenarios using the *Given–When–Then* pattern. These scenarios serve as both living documentation and automated acceptance tests that bridge the communication gap between business experts and developers. While TDD mainly supports developer-centric unit-level verification, BDD operates at a higher level of abstraction, integrating acceptance criteria directly into the development process. [13]

Both approaches aim to shift the role of testing from a late-stage activity to a driving force of the development process itself. Within this agile progression, testing becomes a central design mechanism rather than a post hoc validation step. By continuously intertwining specification, implementation, and verification, TDD and BDD exemplify the paradigm shift from reactive defect detection toward proactive Quality Assurance.

2.3 Software Testing

Software Testing is one of the central activities in Software Engineering. It serves as the primary mechanism for verifying and validating that a system behaves as intended and satisfies its specified requirements. In the most fundamental sense, testing is the process of executing a program with the intention of identifying defects and evaluating whether it performs according to its expected functionality and quality attributes. Both Sommerville and Pressman emphasize that testing is not merely a technical step at the end of development but an integral and systematic part of the Software Development Life Cycle that ensures dependability, quality, and user confidence. [27, 34]

Pressman conceptualizes testing within a broader framework of software quality management. He defines it as a structured, planned, and measurable process that verifies software functionality at different levels of abstraction. Testing, in his view, proceeds through a layered strategy: beginning with unit tests that examine individual components, continuing with integration tests that assess interactions between modules, and culminating in system and validation tests that ensure the complete software product meets its requirements. Each layer contributes to progressively uncovering errors and building confidence in the system. This closely resembles the layered testing of the V-Model (Section 2.2), which progresses from testing the smallest parts of software to the whole Program under Test, broadening the scope of testing gradually. Pressman therefore regards testing as an engineering discipline in its own right, characterized by systematic planning, defined processes, quantitative measurement, and continuous improvement. [27]

Sommerville, on the other hand, situates testing within the life cycle of software validation, focusing less on the mechanics of testing and more on its purpose in ensuring dependability and stakeholder confidence. He distinguishes between development testing, release testing, and user testing, each corresponding to different stages in the maturity of the system. For Sommerville, testing is part of the wider assurance process that demonstrates that a system is "fit for purpose" and behaves reliably under realistic conditions. [34]

Test Suites Within the broader context of Software Testing, the test suite represents the operational foundation of the testing process. As described, both Sommerville and Pressman emphasize that testing is not an isolated activity but an integral part of the Software Development Life Cycle, aimed at verifying that software conforms to its specification and behaves reliably in practice. Pressman describes a test suite as the structured collection of test cases designed and executed to reveal defects and confirm compliance with specified requirements. Sommerville frames the test suite as the practical mechanism through which validation is achieved, linking it directly to software dependability and user confidence rather than to mere defect detection. In both perspectives, the test suite serves as the engine of the verification process. It realises requirements into executable checks and provides systematic evidence for software correctness and reliability. [27, 34]

Test suites comprise multiple test types, referring back to the levels of the V-Model (Figure 2b). Unit testing covers focal methods or other components of the PUT, breaking down the functionality to the lowest code level. Since internal functionality, architecture, or the flow of data inside the PUT is addressed directly, it is classified as white box testing. Integration testing focuses on the interactions of multiple modules or components "to ensure that they work correctly as a system." [36, p. 3] The entire PUT is tested as one in system testing. This ensures that all internal parts and components work together seamlessly, allowing the system to integrate smoothly into the desired environment. Lastly, acceptance testing is the last layer of testing. It involves the PUT as a Blackbox. Often, this testing is not done by the developers themselves, but rather by the customer or user of the system, and includes covering non-functional requirements against the Program under Test. [36]

A good test suite is not simply large or comprehensive; it must also be effective. For both authors, it must be systematic, traceable, and maintainable. Pressman stresses that effectiveness depends on the suite's ability to detect faults efficiently, cover critical paths, and evolve alongside the Program under Test. He highlights attributes such as completeness, repeatability, and maintainability as essential for ensuring that testing remains a controllable and measurable engineering process. A test suite's quality must also be assessed by its purpose, following Sommerville. It should raise confidence in the software's dependability by demonstrating that it performs correctly in realistic operational scenarios. Thus, quantitative adequacy must be complemented by qualitative suitability. [27, 34]

Test Suite Effectiveness Empirical research supports and refines these theoretical foundations. Zhang et al. note that effectiveness remains the central metric in test suite assessment. Effectiveness is multidimensional and involves not only structural coverage metrics such as statement or branch coverage. Semantic dimensions such as fault detection capability, redundancy, and the meaningfulness of assertions are also considered within the definition of effectiveness. The authors emphasize that effectiveness metrics must be interpreted with care since high coverage does not necessarily imply high fault detection ability or reliability, echoing Sommerville's emphasis on validation as confidence building rather than coverage maximization. [34, 40]

Tran et al. identify in their large-scale survey across 354 software professionals [35] fault detection, maintainability, reliability, usability, and coverage as the most important quality attributes of test suites in practice. These attributes collectively capture both quantitative and qualitative aspects of test quality. Quantitative attributes like fault detection and coverage describe the measurable effectiveness of the suite, while qualitative attributes like maintainability, usability, and reliability reflect its practical value in long-term development and continuous integration environments. They

also highlight key challenges that hinder achieving high-quality test suites. Namely, inadequate definitions of quality attributes, a lack of consistent metrics, and missing review and support processes result in inadequate test suites. These findings underline the gap between theoretical adequacy criteria and the operational realities faced by testing practitioners. [35]

Chekam et al. deepen the empirical understanding of quantitative effectiveness metrics by examining the relationship between coverage and fault revelation. Their large-scale study demonstrates that while statement and branch coverage are commonly used indicators, they correlate with fault revelation only when high levels of coverage are reached. In particular, strong mutation testing, measuring a suite's ability to detect deliberately injected faults, shows the most consistent relationship with actual fault detection. This work challenges the *clean program assumption*, emphasizing that coverage alone cannot be relied upon to represent effectiveness and must be complemented by mutation-based adequacy metrics that capture the test suite's true defect detection potential. [8]

Namin and Andrews reinforce the multifaceted nature of effectiveness in [23] and demonstrate that both test suite size and code coverage independently influence effectiveness, but that neither alone provides a complete picture. Their controlled study shows that while larger suites tend to detect more faults, the benefit diminishes when coverage redundancy increases, thus indicating that test quality, not mere quantity, is decisive. Zhang and Mesbah extend this view by examining assertions as a key internal determinant of suite performance in [41]. Their large-scale analysis across five real-world Java projects reveals that the number and coverage of assertions are strongly correlated with test suite effectiveness, even when test suite size is held constant. Therefore, the presence of meaningful, well-distributed assertions has stronger predictive power for fault detection than structural coverage alone. They define assertion coverage as the fraction of code directly checked by assertions and show that it is a more sensitive indicator of effectiveness than statement coverage. Additionally, the type of assertion can significantly affect a suite's ability to reveal faults. This finding directly supports the notion that "coverage without checking for correctness is meaningless", underscoring the need to combine structural metrics with semantic adequacy (i.e., qualitative measures) in evaluating test quality. [23, 41]

Synthesizing these perspectives, a good test suite can be defined as a structured, maintainable, and semantically rich collection of test cases that maximizes fault detection and behavioral verification while minimizing redundancy and maintenance effort. Its effectiveness does not lie merely in executing many lines of code but in the capacity to generate trustworthy, reproducible evidence of correctness through meaningful assertions and comprehensive checking. Methodologically, the assessment of test suite quality should therefore balance objective indicators with qualitative attributes, ensuring that testing supports both technical correctness and long-term dependability of the software system. [23, 27, 34, 35, 40, 41]

***RQ-2.1*: Which quantitative metrics characterize good test suites?**

Quantitative metrics that characterize good test suites primarily measure their fault detection capability and structural adequacy. Common indicators include statement and branch coverage, which quantify how much of the Program under Test code is exercised, and mutation scores, which reflect the suite's ability to detect injected faults. Test suite effectiveness as a measure of a good test suite itself is influenced by its size (i.e., number of tests), which must also be balanced against redundancy to ensure efficiency [23]. Furthermore, assertion quantity, type of assertions, and assertion coverage have been shown to strongly correlate with fault detection and serve as more sensitive predictors of effectiveness than coverage alone [41].

***RQ-2.2*: Which qualitative criteria need to be evaluated for a good test suite?**

Qualitative criteria for evaluating a good test suite focus on its practical usability, maintainability, and contribution to long-term software quality rather than on numeric coverage alone. Key attributes include maintainability, ensuring that tests remain understandable and adaptable as the system evolves, and readability and clarity, which enable effective collaboration and review by developers. Traceability supports systematic validation and accountability [27, 34]. Studies such as [35] highlight usability, reliability, and realism of test scenarios as essential for building developer confidence and ensuring relevance in continuous integration contexts. Together, these qualitative dimensions ensure that test suites not only detect faults effectively but also sustain software dependability, consistency, and ease of evolution over time.

2.4 Software Testing Life Cycle

The Software Testing Life Cycle (STLC) defines a systematic approach to realise Software Testing. Divyani Shivkumar Taley defines the STLC in [11], as seen in Figure 3. It aims to provide assurance that the developed code aligns with the expected behavior and underlines that Software Testing itself is comprised of multiple phases that are codependent on each other. They show the amount of work contributing towards creating meaningful tests. [11]

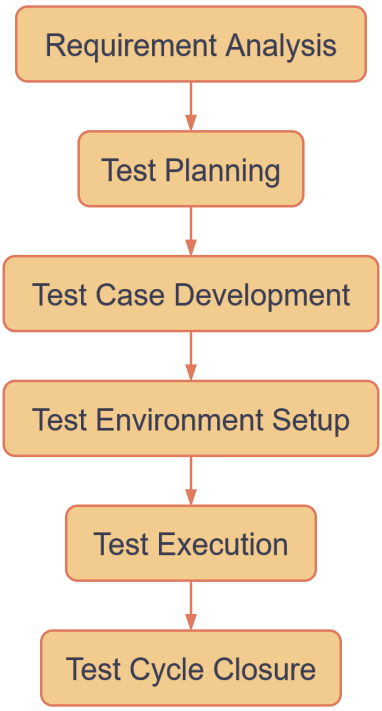


Figure 3: Software Testing Life Cycle [11, p. 819]

Requirement Analysis The first stage focuses on a thorough review of the requirement specifications. The Testing and Quality Assurance (QA) teams analyse the documentation to gain a clear understanding of what must be verified. Whenever ambiguities arise, testers communicate directly with stakeholders to resolve uncertainties. The outcome of this phase is a well-defined scope of testing and an initial prioritization of testing objectives. Early engagement at this stage reduces the likelihood of overlooking critical functionality. [11]

Test Planning Based on the analysed requirements, a comprehensive test plan is developed. This includes defining the test strategy, estimating effort, costs, and timelines, and assigning roles and responsibilities within the testing team. The planning phase also determines the types of testing to be conducted (e.g., functional, performance, or security testing) as well as the required test environment. A solid plan ensures resource optimization and serves as a roadmap for subsequent activities. [11]

Test Case Development During this phase, the testers design detailed test cases and scripts that trace back to the requirements. These test cases specify inputs, preconditions, execution steps, and expected results. To ensure completeness and accuracy, peer reviews or verification activities are typically conducted. If the test environment is available, corresponding test data is also prepared. This step is crucial for achieving high test coverage and ensuring reproducibility of results. [11]

Test Environment Setup The test environment defines the hardware, software, and network conditions under which the tests will run. Although testers document the requirements for this setup, the actual configuration is generally performed by developers or system administrators, often with input from the customer. A properly configured environment ensures that the tests mimic real-world conditions, thereby increasing the validity of the results. [11]

Test Execution Once the environment and test cases are ready, the execution phase begins. Testers systematically run the designed test cases, recording the outcomes. Any deviations from expected behavior are reported as defects, which are then addressed by the development team. After fixes are applied, the testers perform regression checks to confirm the resolution. The cycle of reporting, fixing, and retesting continues until the product aligns with the specified requirements. [11]

Test Cycle Closure The final stage of the STLC involves evaluating the overall testing effort. The team compiles metrics such as the number of executed test cases, defect densities, and their severity distributions. A closure report is prepared, summarizing the quality of the software, coverage of requirements, and outstanding risks or limitations. Lessons learned can be documented to improve future projects. This phase ensures transparency for stakeholders and formally concludes the testing process. [11]

3 Related Work

Software Engineering has continuously evolved towards a mature engineering discipline guided by defined processes, systematic testing, and quantitative evaluation. As emphasized by Pressman and Sommerville, this maturation was driven by the increasing complexity, size, and criticality of software systems and the need for reliable methods ensuring their correctness, maintainability, and quality. [27, 34]

In recent years, the rapid progress of Artificial Intelligence (AI) and particularly Large Language Models (LLMs) has reshaped the Software Engineering landscape. Belzner, Gabor, and Wirsing state that the application of LLMs "promises to support developers in a conversational way with expert knowledge along the whole software lifecycle" [3, p. 3], marking the latest transformative shift in Software Engineering. Their work outlines several key areas in which LLMs are expected to become integral components of future Software Engineering processes. In *Requirements Engineering*, they can assist in eliciting and structuring stakeholder needs by transforming unstructured natural language descriptions into formalized specifications or user stories. They may also detect ambiguities and inconsistencies, thereby improving requirement quality early in the lifecycle. [3]

Within *System Design*, LLMs may generate initial architecture drafts, propose design patterns, and visualize component interactions. Their ability to reason over existing repositories and documentation enables them to recommend modular structures or interface definitions consistent with established best practices and prior projects. [3]

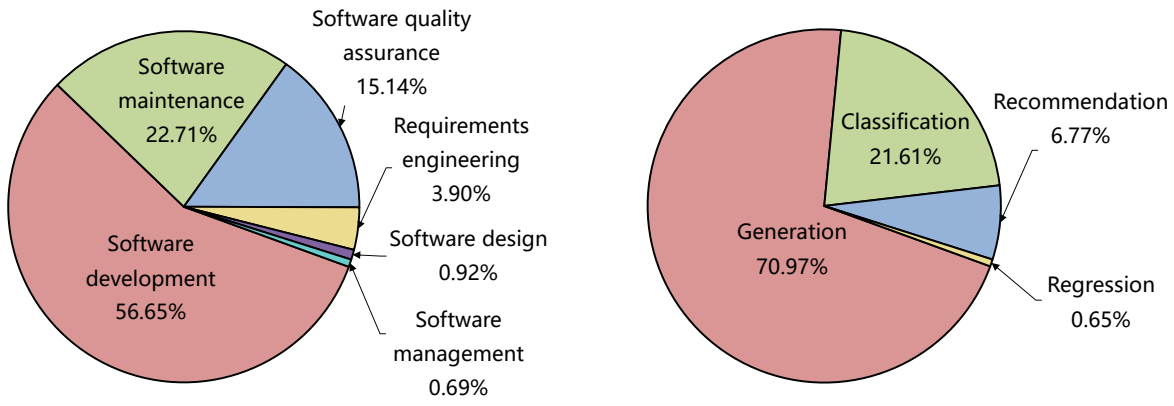
In *Code Generation*, LLMs already demonstrate strong capabilities by translating requirements and design artifacts into executable implementations. Beyond generating isolated functions, they can synthesize coherent modules, integrate dependencies, and refactor legacy code bases, effectively acting as autonomous programming assistants embedded in the development environment. [3]

Finally, in Quality Assurance and Software Testing, LLMs can automate the creation of test cases, predict potential failure points, and interpret test outcomes. They may further support verification by explaining code behavior in natural language or by generating property-based or mutation tests. Through such integration, LLMs could evolve into continuous validation agents that enhance reliability and accelerate feedback throughout the entire Software Development Life Cycle. [3]

3.1 Large Language Model for Software Engineering and Software Testing

This section reviews relevant literature across the Software Engineering and Software Testing domains, providing an overview of current LLM capabilities within these fields.

***Large Language Models for Software Engineering* (2024)** Hou et al. provide a comprehensive systematic literature review of Large Language Models for Software Engineering (LLM4SE) covering studies published between January 2017 and January 2024 [18]. Their analysis reveals that research on LLMs for Software Testing remains comparatively underrepresented within the broader LLM4SE domain. Out of a total of 395 identified publications, only 23 (approximately 5.8%) address testing-related topics. Among these, four focus on testing automation, seventeen on test generation and two on Graphical User Interface testing [18, p. 27].



(a) Distribution of LLM usages in SE activities

(b) Problem classification based on collected studies

Figure 4: Distribution of LLM utilization across different SE activities and problem types [18, p. 26]

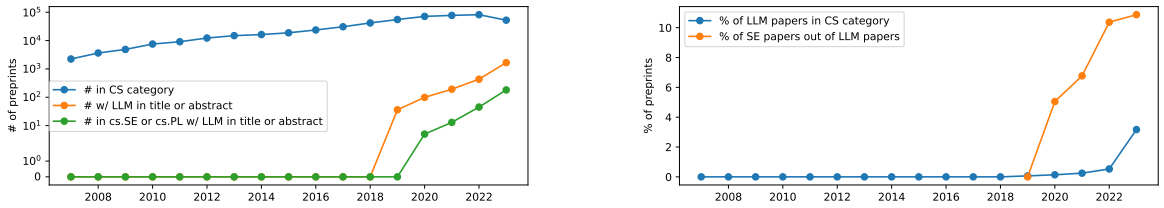
Figure 4a provides an overview of the distribution of LLM4SE studies across different Software Engineering activities. The majority of work focuses on software development and maintenance, whereas only a minor share relates to software quality assurance, where testing research is located. Figure 4b further classifies the identified studies by task type, indicating a clear dominance of generation-based applications, while regression and recommendation tasks play a subordinate role.

The limited proportion highlights that, despite the rapid adoption of LLMs across various Software Engineering tasks such as code generation, documentation, and maintenance, their systematic application to Software Testing is still emerging. The relative scarcity of research in this area underlines both the novelty and the potential impact of LLM-based testing approaches, particularly as models continue to evolve in reasoning and code synthesis capabilities.

***Large Language Models for Software Engineering* (2023)** Further deepening the perspective, Fan et al. provide an extensive survey of the emerging field of LLM4SE and outline a comprehensive agenda of open research challenges [12]. Their analysis confirms the rapidly growing attention of the computer science community towards LLMs and artificial intelligence, with an exponential rise in related publications since 2019 (Figure 5b). This trend illustrates the increasing relevance of LLM-based methods for core Software Engineering activities such as code generation, debugging, and testing.

A key insight of their study is that Software Engineering offers a uniquely automatable environment for evaluating LLM outputs, as program execution provides an objective ground truth for correctness checking. Yet, the authors emphasise that the *Oracle Problem* remains central, particularly in light of the hallucination tendencies of LLMs. While execution can confirm behavioral equivalence, it cannot guarantee semantic validity. To address this, they introduce the notion of an *Automated Regression Oracle*, which uses the existing program behavior as a benchmark for

subsequent adaptations. This approach supports automated regression testing but also risks "baking in" existing faults when applied without additional semantic safeguards.



(a) Trends in number of arXiv preprints

(b) Proportions of LLM papers and SE papers about LLMs

Figure 5: Trends of arXiv preprints and proportions of LLM and LLM-centered papers [12, p. 3]

In the domain of testing, Fan et al. observe that most existing studies focus on the generation of entirely new test cases rather than the augmentation or regeneration of existing suites. They argue that future work should explore how partial oracle information, such as behavioral similarity between new and prior tests, can guide refinement and improve efficiency. At the same time, they highlight the uncertainty inherent in evaluating automatically generated tests. A passing test may simply confirm an incorrect system behavior, whereas a failing one could either reveal a genuine fault or merely contain an erroneous assertion. In the worst case, such misaligned assertions may solidify faulty behavior and obstruct later fixes; conversely, excessive false positives can impose significant verification overheads. These observations underline the necessity of confidence estimation, self-checking mechanisms, and robust evaluation pipelines.

The survey further notes that LLMs enable new areas of testing previously difficult to automate, such as GUI testing, and can enhance established techniques including mutation testing. The latter is identified as a particularly promising direction as fine-tuned models may generate more realistic mutants that mimic developer-like faults, thereby improving adequacy assessment. Therefore, LLMs could themselves act as mutation engines, proposing meaningful variations of program behavior to stress-test both code and its corresponding tests.

Overall, Fan et al. conclude that LLM-based Software Testing remains an early but rapidly maturing field. Its progress depends on combining generative reasoning with empirical validation and on closing the methodological gap between test creation, execution feedback, and correctness assurance, that being an open challenge that continues to shape current research in automated testing with LLMs.

***Software Testing with Large Language Models* (2024)** Complementary to the previous studies, Wang et al. conduct an extensive review entitled *Software Testing with Large Language Models: Survey, Landscape and Vision*, which systematises the state of research across 102 publications and provides a forward-looking perspective on the evolving role of LLMs in testing [36]. Their results indicate that LLMs have been applied to virtually all stages of the Software Testing process - from test design and generation to bug fixing and regression testing - reflecting a rapidly diversifying research landscape (Figure 6).

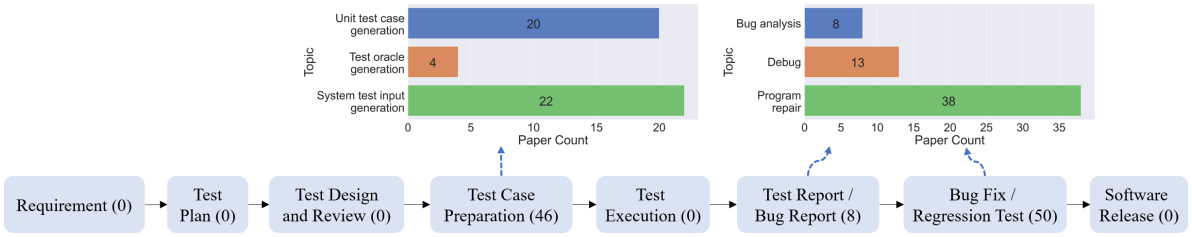


Figure 6: Distribution of testing tasks with LLMs [36, p. 7]
(aligned with Software Testing Life Cycle,
the number in bracket indicates the number of collected studies per task,
and one paper might involve multiple tasks)

In contrast to earlier surveys, which primarily emphasised test generation, Wang et al. highlight that the use of LLMs now spans all major testing activities of the Software Testing Life Cycle. The authors also identify the growing adoption of iterative refinement paradigms such as ChatTester [38], which employs a *validate-and-fix* cycle to improve the correctness and coverage of generated tests. Such strategies represent a shift towards closed feedback loops in LLM-driven testing, where execution outcomes are reintegrated into the generation process for further optimisation.

Nevertheless, the survey emphasises that achieving high coverage remains an open challenge. Increasing the model's sampling temperature, although increasing output diversity does not necessarily translate into broader behavioral exploration and can even lead to instability in generated tests [36, p. 17]. Similarly, the persistent *Test Oracle Problem* continues to limit the reliability of automated evaluation. While LLMs can effectively expose crash-inducing faults through execution, this approach "restricts the potential of utilizing the LLMs for uncovering various types of software bugs" [36, p. 18]. In other words, crash detection alone fails to ensure comprehensive behavioral correctness, especially for subtle logic or semantic defects.

Wang et al. further discuss practical limitations stemming from the computational requirements of large models. Many academic studies rely on medium-sized variants due to cost constraints, which can impede reproducibility and result in significant performance variation compared to frontier-scale systems [36, p. 19]. This highlights a methodological challenge in maintaining consistent benchmarks for evaluating LLM-based testing approaches across research environments.

Looking ahead, the authors identify several promising directions for future work. They advocate for expanding the role of LLMs into the earlier phases of Software Testing, like requirements definition, planning, and test design, as well as into higher levels, including integration and acceptance testing, as currently, most research focuses on creating unit tests or generating input for system tests. This call for broader coverage aligns with the vision of a holistic, model-driven testing workflow that bridges natural-language specifications and executable verification artifacts. Similar to observations made by Santos et al. in [29], such extensions could enable end-to-end automation from requirements engineering to validation.

Overall, Wang et al. portray a research area transitioning from isolated proof-of-concept studies towards integrated, feedback-driven testing ecosystems. The convergence of generation, execution, and repair phases within LLM-based testing frameworks signals a paradigm shift towards more adaptive and self-correcting testing methodologies, laying the conceptual foundation for the subsequent discussion on test case generation.

3.2 Automated Test Case Generation with LLMs

***ChatUniTest* (2024)** Chen et al. introduce *ChatUniTest* in [9]. This "Framework for LLM-Based Test Generation" creates unit tests for individual methods of a PUT. ChatUniTest follows the *generation-validation-repair* approach, visible in Figure 7. Each method of the PUT is extracted and integrated into a prompt for the generation of appropriate unit tests. The resulting tests are parsed from the LLM response and undergo validation for syntactic correctness, successful compilation, and runtime executability. Any failure during this validation will trigger an attempted repair of the test, firstly through a rule-based approach, followed by an LLM-based repair. Notably, the tests are only repaired for technical errors. The assertion outcome (either pass or fail) is not the subject of ChatUniTest's *validation-repair* loop. Chen et al. highlight a line coverage of 59.6% that was achieved "across diverse projects" [9, p. 4].

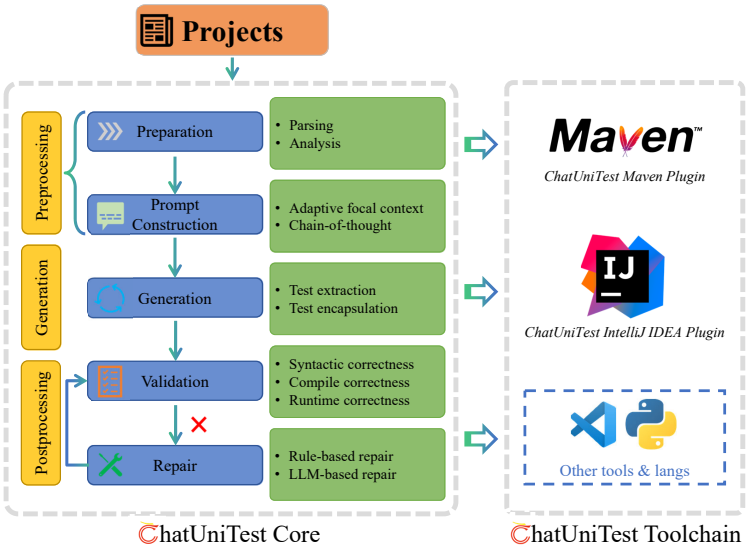


Figure 7: ChatUnitTest Overview [9, p. 2]

The paper fundamentally highlights that (a) LLMs are capable of generating unit tests and (b) can provide valuable repair help where rule-based repair methods fail. The paper fails to provide any insights into the effectiveness of generated test cases and solely relies on line coverage as the optimization objective. It does not assess the semantic correctness or fault-revealing capability of these tests. Therefore, its contribution towards true automated test case generation overall is marginal. ChatUniTest should be viewed as a tool for initial unit test generation, where the unit tests themselves will be subject to further development by humans based on their experience, the execution results, and the testing targets of the project. [9]

***No More Manual Tests?* (2024)** Yuan et al. use a dataset of 1,000 Java focal methods. Against this dataset, unit tests are generated. As a baseline, they use ChatGPT without adaptations and find that all of the generated tests are syntactically correct while only 39.0% compile correctly. Of all tests, only 22.3% are passing the execution (see Table 7 in [38, p. 17]). To enhance this performance, Yuan et al. propose *ChatTester* in [38].

ChatTester follows an iterative repair strategy. As seen in Figure 8, ChatTester splits the initial test generation into two parts. The intention of the focal method is described by ChatGPT in natural language in the first step. This output, in addition to the focal method and the instruction to generate a unit test based on this data, is used in the second step. If the returned test compiles, no refinement is done, and the test is assumed to be final. Any compilation error will invoke the *Iterative Test Refiner* part of ChatTester. For this, any error messages as well as contextual code information is combined into a new prompt. A new test is generated using this prompt and will follow the same validation structure. This *validation-repair* cycle is repeated until either the test compilation is successful or the maximum number of refinement iterations is reached; both leading to a stopping of the refinement process. The last state of the test will be considered valid and output by ChatTester. [38]

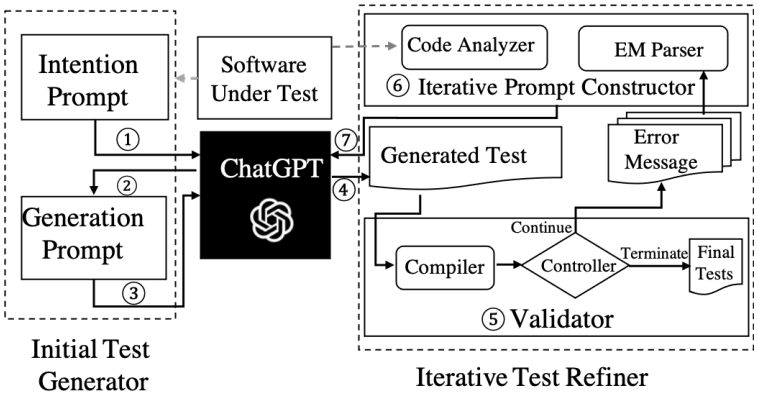


Figure 8: ChatTester Overview [38, p. 13]

Using the ChatTester approach, Yuan et al. show a syntactical correctness of 100% equal to the initial output of ChatGPT without enhancements while increasing the number of compilable tests to 73.3%. The tests with a successful execution nearly double to 41.0%. With the lower number of repair loops (set to max. three iterations in [38]), Yuan et al. underline the strength of LLM-based repair and repeated reprompting. Similar to ChatUniTest [9], no attempt at analysing assertions was made. They explicitly state that they "only focus on fixing compilation errors instead of execution errors [i.e., failed assertions], since in practice it is challenging to identify whether a test execution failure is caused by the incorrect test code or by incorrect focal methods" [38, p. 15].
Two research results can be drawn from this paper:

(a) The correction of syntactical errors is not considered in the approach at all. It is shown that all generated tests have a correct syntax, initially as well as after any refinement. This can be attributed to improved LLM performance on coding tasks by newer models.

(b) Reprompting with additional information from the environment helps the LLM improve its unit tests significantly. The work shows that the error messages with some contextual information are a sufficient input for the LLM to correct compilation errors nearly doubling the number of compilable tests. [38]

***Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing* (2023)** Dakhel et al. attempt to include assertion correctness for unit test generation in their

approach Mutation Test case generation using Augmented Prompt *MuTAP* [10]. After an initial LLM-based test generation, any syntax errors are fixed by the LLMC (LLM component) with a repair prompt. Any persistent syntax errors are resolved through the removal of the affected lines of test code. No natural language descriptions of the method's intent are included, such that the LLMC needs to infer the intended behavior of the function. Possible incorrect oracle assumptions may lead to failed assertions. Through the execution of the test, any incorrect assertions are detected and MuTAP replaces the LLM-generated oracle of those assertions with the observed PUT output. This results in a so-called "Initial Unit Test" (IUT) that only includes passing assertions. Figure 9 shows that these IUTs are then assessed for their quality and effectiveness using mutation testing - the core novelty of MuTAP. [10]

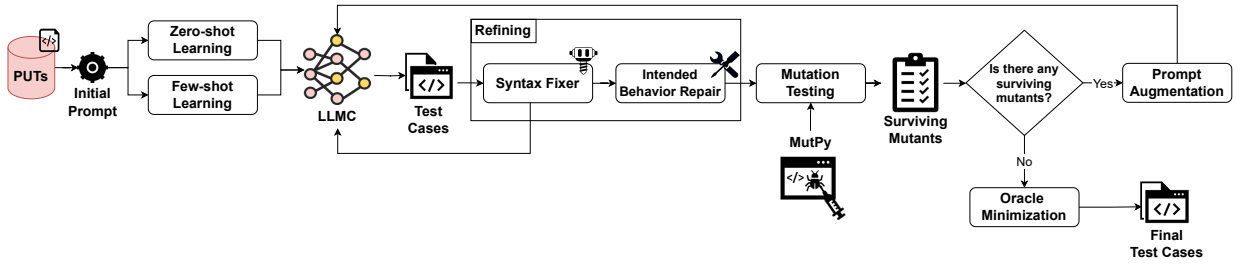


Figure 9: MuTAP Overview [10, p. 4]

The mutation testing introduces bugs into the PUT and asserts whether set mutants are detected by generated unit tests. If the number of detected mutants is not equal to the total number of injected mutants, the unit test is evaluated as incomplete and needs to be refined. This refinement consists of a new prompt that involves the current unit test, the undetected mutants, and the instruction to *"Provide a new test case to detect the fault in prior code"* [10, p. 6]. The reprompted unit test then undergoes the described process seen in Figure 9 starting from the *Syntax Fixer*. If a unit test successfully killed all mutants, no further iterative changes are done. Possibly redundant assertions are reduced through a Greedy algorithm based on the mutants killed by the individual assertions. The reduced unit test is considered final and represents the output of MuTAP. [10]

Dakhel et al. highlight that around "60% of the test cases generated by Codex encounter compilation issues due to syntax errors" [10, p. 13]. This contrasts the findings of Yuan et al. but can be attributed to the selection of the Codex LLM instead of ChatGPT. Their refining process before applying the mutation testing is therefore justified, but its need for later, more capable models remains to be investigated. Importantly, they show that LLMs are capable of correcting assertion behavior based on real PUT outputs. MuTAP assumes a correct PUT on which all assertion failures are due to incorrect oracle predictions. This, in combination with the mutation testing, enhances the effectiveness of the tests. Dakhel et al. stress that a good mutation score[3] does not directly correlate to other quality metrics like coverage and is only weakly linked to the test's ability to detect faults. [10]

[3]Mutation score is defined as the number of killed mutants over the total number of mutants.

***TestART* (2025)** The generalized process of LLM-based unit test generation, including repair loops, is described by Gu et al. in [15]. They characterize the workflow (Figure 10) as mainly consisting of three components: the prompt constructor, test validator, and processing engine.

The task of the prompt constructor is to build a generation instruction, typically by initializing the system prompt and combining the source code and other information into the following prompts. The LLM is the generating test cases, which are analysed by the test validator. This validation can include syntactical or assertion correctness. Following that, a repair or refinement is attempted that may include static analyses, error message parsing, or LLM-based repairs. [15]

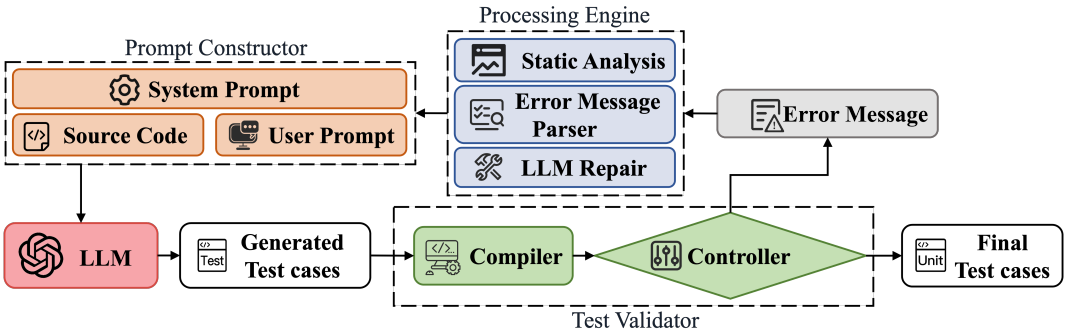


Figure 10: Common workflow of LLM-based unit testing generation method [15, p. 3]

Gu et al. themselves implement the described workflow in their approach called *TestART*. It includes pre-processing before the initial generation to reduce the token count and possible hallucinations. The generated tests are syntactically repaired and corrected for incorrect assertions in a dedicated repair loop, effectively removing or correcting all non-compilable or failing tests. The remaining tests are then analysed for their coverage behavior. Using JUnit and OpenClover, the code coverage achieved by the test is identified. If coverage standards are not yet met, the test case will be refined based on a new prompt that includes the current coverage report and stated coverage goal. This iterative process is repeated until sufficient coverage is provided by the returned test cases. [15]

TestART therefore optimizes for coverage as a metric of test case quality. Notably, this differs from MuTAP [10] which uses the mutation score as a quality stand-in. In tests, TestART achieved around 70% line and branch coverage with a pass rate[4] of over 71%. This shows that LLM-based generation is not only limited to pure unit test generation with a focus on syntactical correctness and repair abilities to enforce correct assertions. The current LLM capabilities, as proven in [15], also include the optimization of test cases towards quality metrics like the mutation score or coverage. [15]

[4]Pass rate is defined by Gu et al. as "the percentage of test code that is syntactically accurate, compiles and runs without errors or failures" [15, p. 12].

***RQ-1.1*: Can LLMs provide or approximate suitable test oracles?**

Evidence from current research suggests that LLMs can only approximate suitable test oracles to a limited extent. While program execution provides an automatable ground truth [12], the persistent *Oracle Problem* and lack of semantic reasoning hinder reliable validation. Approaches such as MuTAP [10] and TestART [15] approximate behavioral oracles by aligning assertions with observed program outputs, implicitly assuming the correctness of the PUT. This risks reinforcing faulty behavior. As highlighted by Wang et al. and Fan et al., current LLMs cannot ensure differentiate faulty code from faulty tests, leaving true oracle capability unresolved.

Empirical Evaluations and Complementary Approaches Beyond the dedicated frameworks discussed above, several empirical and domain-specific studies have evaluated or extended the effectiveness of LLM-based unit test generation. Schäfer et al. present *TESTPILOT*, an empirical evaluation of iterative loop–based repair strategies similar to those employed by ChatUniTest and ChatTester. Their results indicate that such iterative correction can achieve a mean line coverage of 70.2% and branch coverage of 52.8%, thereby approaching the levels reported for TestART. TESTPILOT further highlights the substantial performance variance between individual Large Language Models, underlining that test generation quality remains strongly model-dependent even when following identical prompting pipelines. [30]

Siddiq et al. conduct a complementary empirical study using OpenAI's Codex to generate JUnit tests for the HumanEval and EvoSuite SF110 benchmarks. While achieving up to 80% line coverage on HumanEval, the approach drops to merely 2% on the more complex EvoSuite SF110 dataset, accompanied by a low execution success rate and a reduced number of generated tests relative to the available focal functions. These findings emphasise that model generalisation and domain complexity critically affect both coverage and executability of LLM-produced tests. [33]

Finally, Shin et al. propose *Domain Adaptation for Code Model-based Unit Test Case Generation*, which employs smaller, fine-tuned models rather than frontier-scale LLMs. By adapting the model to project-specific data, the authors report improvements of more than 15% in both line coverage and mutation score compared to untuned baselines. Although this work does not implement iterative repair loops, it demonstrates that domain adaptation can partially compensate for limited model capacity and resource constraints, suggesting a complementary path to large-model–centric generation pipelines. [32]

***RQ-1.2*: Can LLMs generate syntactically correct and executable test code?**

Research shows that LLMs can generate syntactically correct and executable test code, with success depending on model capability, the selected dataset, and refinement strategy. ChatUniTest [9] and ChatTester [38] achieve high syntactic validity, raising passing executions from 22.3% to 41.0% through iterative repair. TestART [15] attains pass rates above 70% via dedicated syntax correction. Empirical studies [30, 33] confirm that residual failures largely stem from unresolved dependencies or environmental misconfiguration. With validation and repair, LLMs reliably produce compilable tests.

***RQ-1.3*: Can LLMs understand the structure of the Program under Test, make correct references in the test code, and handle interactions with the working environment?**

LLMs demonstrate partial understanding of a Program under Test's structure and can generate test code with correct references when contextual information is explicitly provided. ChatUniTest [9] and ChatTester [38] achieve accurate method calls through prompt guidance, while MuTAP [10] and TestART [15] encounter frequent dependency and import errors without preprocessing. Empirical studies [30, 33] show many execution failures stem from unresolved dependencies or framework issues rather than syntax. Complex, strongly typed languages amplify these effects. LLMs thus are capable of capturing local structure but seem to lack a consistent understanding of global architecture and runtime environments.

***RQ-1.4*: Can LLMs produce usable tests in a single shot or do they require iterative human/feedback guidance to generate runnable tests?**

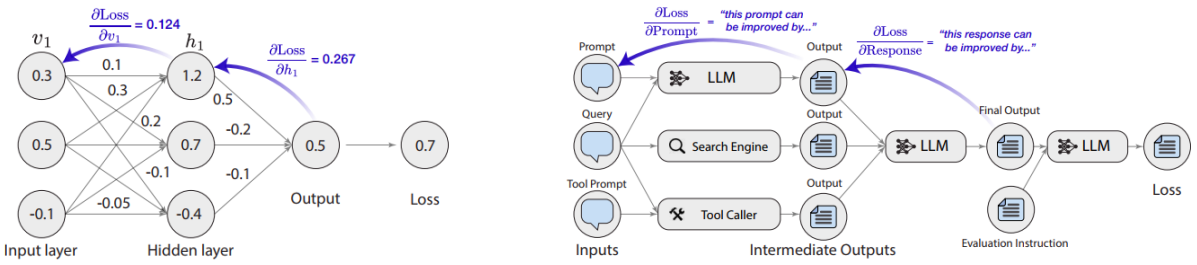
Studies consistently indicate that LLMs rarely produce fully runnable tests in a single generation step. This is in line with content generation of any type by LLMs being improved by iterative refinement [22, 26]. While models can generate syntactically valid code, most outputs require iterative validation and feedback to achieve executability. Frameworks such as MuTAP [10] and TestART [15] explicitly implement *generation–validation–repair* loops, demonstrating substantial improvements in compilation and pass rates through automated refinement. Empirical evaluations [30, 33] likewise confirm that test quality increases significantly after reprompting or environment-based correction. Consequently, usable tests generally emerge through iterative feedback rather than single-shot generation, highlighting the importance of structured refinement mechanisms in LLM-based testing.

***RQ-1.5*: Which levels of testing can LLMs generate and where do their current capabilities reach their limits?**

Current evidence indicates that LLMs are primarily effective in generating unit and, to a lesser extent, system tests. Wang et al. identify only 24 papers addressing unit testing and 22 focusing on system testing, while no studies explicitly target integration or acceptance levels [36]. This distribution underscores that existing approaches concentrate on testing isolated functions or providing test inputs for complete programs. They rarely target interactions between components or user-facing validation. The absence of research on higher testing levels suggests that LLMs might struggle with contextual dependencies and cross-module reasoning, marking a key limitation in extending current capabilities beyond unit-level generation.

3.3 TextGrad

Yuksekgonul et al. propose *TextGrad* in [39], introducing both a conceptual foundation and an implementation framework for optimizing LLMs through differentiation via text. Instead of relying on numerical gradients, TextGrad formulates the optimization process in natural language, effectively treating model feedback as a form of "textual gradient". This analogy extends the principle of backpropagation known from neural networks to the domain of language-based reasoning, allowing complex Blackbox AI systems to be optimized without requiring access to internal parameters. The conceptual correspondence between numerical and textual differentiation is illustrated in Figure 11. [39]



(a) Neural network and backpropagation using numerical gradients

(b) Blackbox AI systems and backpropagation using natural language "gradients"

Figure 11: Automatic "Differentiation" via Text [39, p. 3]

The framework operates on the minimal configuration of two LLMs: a predictor and an evaluator. Given a prompt O and a query Q, the predictor generates a prediction P according to Equation 1. The evaluator then assesses this output using an instruction I and produces an evaluation E as defined in Equation 2. Both components can be instances of the same underlying model, separated only by their contextual configuration, such as system prompts or temperature settings. [39]

$$O + Q \xrightarrow{\text{LLM}} P \quad (+ \text{ denotes string concatenation}) \tag{1}$$

$$I + P \xrightarrow{\text{LLM}} E \tag{2}$$

The optimization target in this setup is typically the prompt O. Based on the evaluator's output E, a local feedback signal F is generated that expresses how the prediction P could be modified to better satisfy the evaluation criteria. This feedback is obtained through an additional LLM query, for example:

```
"Given this Prediction P and this Evaluation E,
how should the Prediction change to improve the Evaluation?"
```

The obtained feedback F is then propagated back to update the optimization variable O, effectively closing the feedback loop. The LLM is again queried to propose an improved version of the prompt:

```
"Given this Prompt O, its use to generate the Prediction P and
the feedback F against P, how should the Prompt be changed so that
the Prediction resulting from the new Prompt would satisfy that feedback?"
```

Through this iterative refinement, TextGrad establishes an emergent form of gradient descent in text space. Each iteration produces a slightly improved prompt, gradually steering the LLM toward higher performance with respect to the evaluation metric. The resulting optimization process can be seen as a linguistic analog to stochastic gradient descent, yet with all intermediate steps expressed in natural language rather than numerical form. [39]

Two classes of optimization problems are distinguished. The first - instance optimization - aims to improve the solution itself, where the generated output (e.g., code, reasoning, or symbolic structure) constitutes the optimization target. The second focuses on prompt optimization, in which the goal is to enhance the general performance of a model across multiple queries or tasks [39, p. 6].

3.4 LLM-as-a-Judge

Zheng et al. propose the *LLM-as-a-Judge* approach in [42], a systematic framework that employs powerful language models such as GPT-4 to evaluate the quality of responses produced by other Large Language Models. The approach aims to replace costly and time-consuming human evaluation with scalable, automated, and explainable judgment models that approximate human preferences. It has been empirically validated against human experts and crowdsourced evaluations, achieving an agreement rate exceeding 80%, comparable to inter-human agreement levels.

Types of Judges Three principal types of LLM-based judges are proposed, each tailored to different evaluation contexts:

- **Pairwise comparison**: The model is presented with a question and two answers and must determine which one is superior or declare a tie. This method captures relative preferences but scales quadratically with the number of compared systems, limiting its applicability in large-scale studies.
- **Single-answer grading**: The model assigns an absolute score to a single response. This variant is more scalable but may be less stable, as absolute ratings can fluctuate more strongly than relative judgments.
- **Reference-guided grading**: A reference solution is provided, allowing the judge to compare responses against a known standard. This is particularly effective for structured domains such as mathematics or programming tasks.

Each variant exhibits specific advantages and trade-offs regarding scalability, interpretability, and sensitivity to contextual differences. They can be combined to strengthen overall evaluation reliability. [42]

Advantages Zheng et al. underline that the LLM-as-a-Judge approach offers two main advantages: scalability and explainability. It drastically reduces the need for human annotators, enabling rapid and large-scale benchmarking across diverse conversational and reasoning tasks. Secondly, unlike traditional metrics based on similarity scores, LLM judges not only produce quantitative evaluations but also textual explanations that provide insight into their reasoning process, thereby increasing the transparency and interpretability of the evaluation. [42]

Limitations Despite its effectiveness, several systematic limitations have been identified. The authors classify these as distinct forms of bias or reasoning deficiencies:

- **Position bias**: The model may favor the first or second response based solely on presentation order. This tendency mirrors known cognitive biases in human judgment and can distort outcomes unless results are averaged or randomized. It can be addressed through multiple judgments of the same answers in different configurations (position-swapping).
- **Verbosity bias**: Longer or more elaborate responses may be incorrectly judged as superior, even when they add no new information.
- **Self-enhancement bias**: Some judges exhibit a slight preference toward responses produced by their own model family, though this effect is minor and varies across models.
- **Limited capability in grading math and reasoning questions**: LLMs often fail to assess mathematical or logical accuracy correctly, even when they are capable of solving the underlying problem themselves. For example, GPT-4 has been shown to misjudge elementary arithmetic tasks by being misled by incorrect responses or to label erroneous reasoning chains as correct despite being able to derive the right solution when queried independently. This limitation stems from the model's contextual dependency as it tends to reproduce or align with the structure of the provided answers rather than verifying them rigorously. Such cases highlight that the model's evaluation competence is still constrained by its reasoning process and prompt framing. [42]

While mitigation strategies such as chain-of-thought prompting, reference-guided evaluation, and position-swapping have been proposed to alleviate these biases, none entirely eliminate them. Nonetheless, given the demonstrated high agreement rate with human judgments and the interpretability of its outputs, the LLM-as-a-Judge paradigm represents a valid and practical approach for scalable evaluation of LLM and other AI-based systems. [42]

***RQ-1*: Are LLMs able to generate test suites at all?**

Recent advances in Large Language Models have established them as powerful tools across Software Engineering, capable of reasoning about source code, generating executable artifacts, and assisting in validation tasks. Their growing integration into SE workflows reflects rapid progress in contextual understanding, code synthesis, and self-correction. Concepts such as *LLM-as-a-Judge* [42], which positions models as evaluators of generated artifacts, and *TextGrad* [39], which introduces gradient-like optimisation through textual feedback, provide the theoretical foundation for using LLMs not only as generators but also as adaptive components within iterative refinement loops. The growing capabilities of LLMs have driven their adoption in software testing as well.

The reviewed literature demonstrates that LLMs can indeed generate executable tests and, by extension, form test suites. Early frameworks such as ChatUniTest [9] and ChatTester [38] confirmed their ability to produce syntactically valid unit tests and to repair non-compiling or failing outputs. More advanced approaches like MuTAP [10] and TestART [15] incorporated behavioral feedback and coverage-guided refinement, achieving pass rates above 70%. These results, supported by empirical evaluations [30, 33], indicate that the generative accuracy of LLMs improves with scale, model evolution, and feedback integration.

The sub-questions of this research collectively underline that LLMs can partly approximate oracles, generate syntactically correct and executable tests, and reproduce local program structures, yet still rely on iterative guidance to produce reliable results. Their competence currently concentrates on unit and system testing, while no approaches currently explicitly target integration or acceptance testing.

In summary, LLMs are demonstrably capable of producing test cases and small-scale suites and their abilities continue to improve. However, to the authors' knowledge, no study to date has attempted to generate full-scale, multi-level test suites covering an entire PUT.

4 LIFT - LLM-based Iterative Feedback-driven Test suite generation

As described in Section 3, there already exists a wide range of research targeting the automated generation of unit tests using LLMs. The fast development of LLM capabilities is visible over the past years. While authors like Dakhel et al. and Chen et al. suggest refinement loops for fixing structural validity (i.e., syntax errors) in [9, 10], later publications like MuTAP [38] or TestART [15] omit explicit syntax repair loops completely, indicating that later LLM generations do not encounter these errors in a significant magnitude anymore. Instead, newer research focuses more on generating higher quality test cases through the optimization of quality representatives.

Yet, all reviewed approaches focus solely on unit test generation. This focus is understandable as unit tests are viewed as the lowest step in the V&V process (see V-Model Figure 2b) and provide a sufficiently complex yet isolated challenge that needs to be addressed as they represent the foundation of any test suite. Their isolated and quantifiable nature makes them the ideal research subject. Datasets like Defects4J [21], EvoSuite SF110 [14], or the HITS dataset [37] provide a well-established baseline against which to test different generation approaches and quantify their effectiveness. Solving this problem is a big step towards test automation. Nevertheless, Software Testing of complete Programs under Test entails more than unit tests. Integration, system, and acceptance testing are defined as the next higher complexity layers. The fixation on unit tests keeps the real-world use limited to this lowest level of testing.
Two complementary research gaps emerge from this review:

(1) the *Scope Gap*, as current LLM-based methods remain restricted to unit-level test generation without addressing higher testing layers such as integration or system testing, and
(2) the *Optimization Gap*, as existing iterative repair strategies are discrete and task-specific, lacking a unified, gradient-like feedback mechanism for continuous improvement of the entire test suite.

This thesis introduces **LLM-based Iterative Feedback-driven Test suite generation (LIFT)**. LIFT aims to close these research gaps and to provide a step towards the automation of test suite generation. It generates whole test suites for complete PUTs instead of individual unit tests for one focal method at a time. Based on the TextGrad [39] and LLM-as-a-Judge [42], LIFT broadens the application of LLMs in Software Testing to include integration and system testing and generate complete test suites. It utilizes the instance optimization of TextGrad and applies it to the suites themselves - not only for fixing incorrect any syntactic structure and test oracles but also for improving the overall quality iteratively. To achieve this, the test suite becomes the object that is continuously enhanced based on feedback of an Evaluation-LLM. The novelty of LIFT lies in the combination of iterative feedback based on a selected metric like coverage or mutation score with LLM capabilities to evaluate those metrics and provide feedback. This gradient-like natural language feedback is used to improve the output test suite as proposed by TextGrad [39] with the enhancement of LLM-as-a-Judge [42] including reasonable improvement steps and judging the suites quality.

4.1 Concept & Architecture

The whole test suite is the deliverable of the LIFT process and the product of instance optimization via text introduced by TextGrad [39] - applied to Software Testing. The suite itself represents the instance variable to be optimized, is completely LLM generated and refined without external input. Feedback from the Test Suite Evaluator provides the "textual gradient" describing how the suite should change to improve its overall performance/effectiveness. Each iteration thus resembles a gradient-descent step performed in natural language space.

The resulting process architecture, illustrated in Figure 12, implements this iterative optimization loop as a sequence of recurring iterations. LIFT performs optimization in three deterministic phases - *Generation/Refinement*, *Execution*, and *Feedback* - which together constitute one iteration. The order and logic of these phases is fixed. The actions taken by each LLM agent are contained within each phase and do not influence the order of execution.

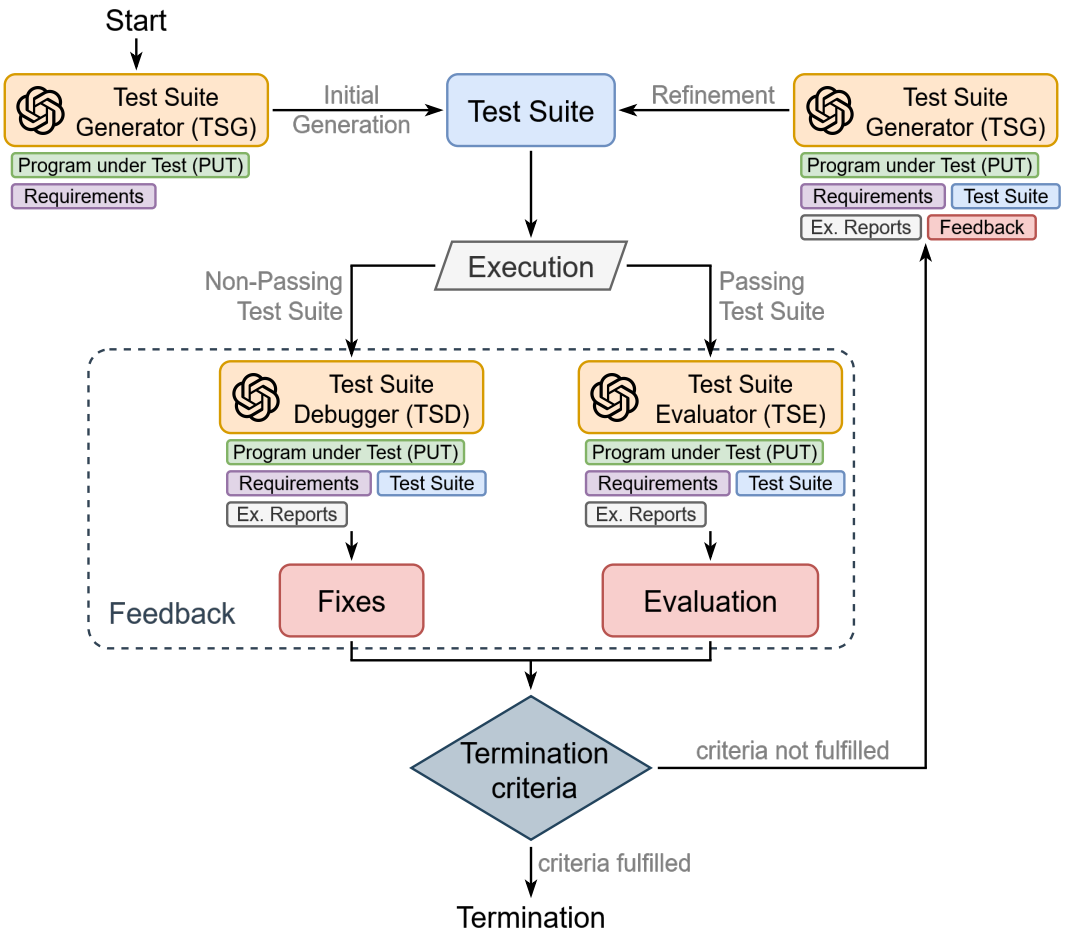


Figure 12: Overview of LIFT

Initital Generation The test suite is initially created from the provided requirements and the Program under Test (PUT) as shown in the upper left-hand corner of Figure 12. The Test Suite Generator (TSG) uses this information to build an initial test suite.

Execution The execution invokes all available tests. The execution report includes not only possible errors and the test results (i.e., passing, skipped, and failed tests) but also the line and branch coverage achieved by the current test suite. Each execution runs all tests and does not include cached information to avoid side effects.

Feedback The type of feedback is decided by the execution outcome. Failing tests or errors in the test suite will result in an analysis of these problems by the Test Suite Debugger (TSD). This analysis will cluster any common root-causes like incorrect import strategies or incorrect program behavior assumptions used throughout multiple tests and suggest solutions.
A fully passing test suite will trigger the evaluation by the Test Suite Evaluator (TSE) that analyses the suite's performance based on achieved coverages and the implemented tests. Its aim is to identify weaknesses like uncovered execution paths, untested edge cases, or integration gaps and suggest a textual gradient encoded in the improvements towards a better test suite.

Termination After the feedback, the iteration concludes with the decision to terminate the LIFT or continue refining the test suite by applying the fixes or improvements from the evaluation. Termination criteria can be that (a) the TSE determines in its evaluation that the current state of the test suite has reached a satisfactory quality and does not need to be enhanced further, (b) certain pre-defined metric thresholds were reached, or (c) the maximum number of iterations was executed.

Refinement The refinement - done by the Test Suite Generator (TSG) with all artifacts generated in the last iteration - is the first step of the next iteration. The Test Suite Generator modifies existing tests or creates new ones based on the textual gradient encoded in the feedback (fixes or evaluation). After each refinement is applied, the temporary artifacts created based on the test suite's state before the update are removed. This includes the execution reports and the feedback files. This is done to ensure data consistency and prevent outdated information from influencing subsequent iterations.

4.2 Agents & Environment

At LIFT's core, three LLM agents interact with the test suite. The Test Suite Generator (TSG) interacts with the test suite directly to create or update tests, the Test Suite Debugger (TSD) suggests fixes for problems in the test suite, and the Test Suite Evaluator (TSE) gives textual feedback to improve the suite's performance.

The separation into three specialized agents follows the principle of modular optimization, mirroring the decomposition of a gradient update into generation, error analysis, and evaluation. This design prevents feedback contamination between stages, ensures reproducibility, and reflects the independent roles of gradient computation and instance update in TextGrad's optimization analogy.

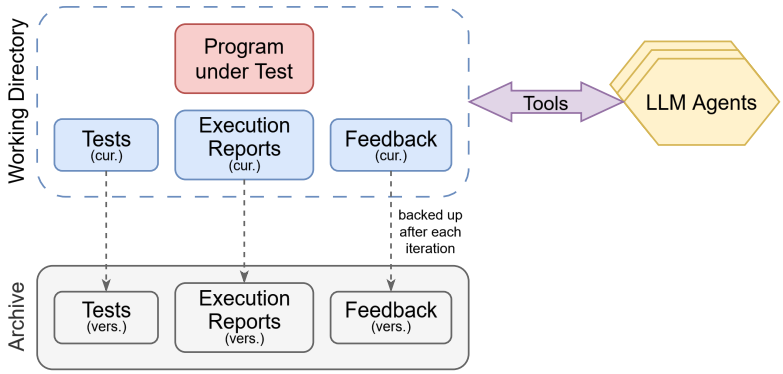


Figure 13: Archiving process of LIFT
(cur.: current; vers.: versioned)

For each iteration, all changes are archived as shown in Figure 13. This includes the created/updated code for the whole test suite, the generated reports of the suite's execution, and the feedback for improvements. The generated execution reports consist of a test execution report (`xml`), a coverage report (`xml`), and a visual report for traceability (`html`). The generated feedback is only available in files (`markdown`) as no direct communication between agents is implemented. Only the most recent data stays available to the agents.

Across all iterations, code coverage serves as the primary quantifiable metric guiding the optimization process. Statement and branch coverage values are extracted from each execution report and provide the numerical basis for evaluating the test suite's effectiveness. These quantitative measures are one cornerstone of the qualitative evaluations generated by the TSE, which additionally assess factors such as assertion quality, completeness, and redundancy. Together, these perspectives allow the system to balance measurable and reasonable coverage growth with meaningful test design improvements.

Each agent instance is stateless to ensure reproducible and isolated optimization steps. For every phase of an iteration, a new agent is instantiated without access to prior context. This design eliminates hidden state accumulation and guarantees that all decisions are based solely on the artifacts explicitly available in the working directory. The LLM's role is defined by the system prompt and its current task by the user instruction given as the first message. No further message output is required by the agents. Information like the fixes or evaluation, therefore, needs to be provided in a file to be available after the completion of the agent's task.

System Prompt Each agent's behavior is defined by a structured system prompt S, which formalizes its role and operational constraints to ensure consistency and reproducibility across iterations. The prompt is composed of the following elements:

- R: the *role* that the model is assuming in answering the instruction. This is a general description of the assumed experience, core skill set, and most important "character traits".

- E: the *environment* that the model operates in. This is reduced to the operating system, information about the programming languages locally installed, and the testing frameworks used.
- W: the *workflow* that the model needs to follow to correctly fulfill the instruction. The workflow guides the model from understanding and retrieving the available data to the needed analyses and the output required.
- O: the *expected output*. This defines which files are allowed to be created/edited and what parts of the project are off-limits.
- D: the *conversation directives* guiding the model not to request any further textual input. It enforces that assumptions shall only be made based on the available information (i.e., the project files).
- P: the *critical principles* reinforcing behavioral patterns for the central workflow steps. This limits what the models should focus on and tightens the scope of their objective.
- A: any optional *additional information* provided. In practice, this is either the pattern to use for test implementations or feedback reporting.

The system prompt for each agent is defined as:

$$S = R + E + W + O + D + P + A \quad (+ \text{ denotes string concatenation}) \tag{3}$$

Tools All LLM agent interactions with the environment are realised through tools. They ensure that all changes are contained to the working directory. No tools for networking interactions like web requests or direct communication between LLM agents are implemented. The following tools are available to all LLM agents:

- `list_dir(...)` - lists all folders and files recursively (with optional parameters for hidden files, start directory or pattern matching),
- `read_file(...)` - reads a file's content from a given byte offset and length and reports whether more bytes are available,
- `write_file(...)` - writes or overwrites files within the working directory (default: no overwrite),
- `delete_path(...)` - deletes a file or directory, and
- `replace_in_file(...)` - replaces the first occurrence of a given string with a new value.

4.2.1 Test Suite Generator

The Test Suite Generator is the central generative agent in LIFT and the only one allowed to modify the test suite directly. It performs the role of the optimizer - updating the test suite based on textual feedback that encodes directions for improvement. The TSG is responsible for creating the initial suite, correcting errors, and refining test coverage and quality during subsequent iterations.

The behavior of the TSG is defined by a static system prompt (Subsection A.1) specifying its role, environment, workflow, output boundaries, and guiding principles. It defines the agent's behavior as a deterministic test-generation expert.

The corresponding workflow is highly structured and divided into six major phases: (1) project discovery, (2) comprehensive understanding, (3) requirements gathering, (4) feedback analysis (conditional), (5) test implementation, and (6) final review. Each phase is expressed in procedural steps that define the agent's internal reasoning and file interaction pattern. To prevent deviation from the prescribed workflow, conversation directives explicitly prohibit further user interaction or open-ended reasoning.
Once the task is completed, the agent terminates automatically, signaling completion with the reserved token `<DONE>`.
Depending on the iteration type, the TSG operates under one of three instruction modes defined by the initial user message:

> **Initial Generation**: Triggered by the instruction *"Generate an initial test suite for the local project* `<projectname>` *based on the given requirements!"*.
> The TSG constructs a complete suite covering all functional requirements defined in the requirements document.
>
> **Error Correction**: Triggered by *"Error(s) during the collection or fail(s) occurred during execution of the test suite for the local project* `<projectname>`*! Please correct the test suite!"*.
> This mode is invoked after the TSD detects and reports syntactic, import, or runtime errors. The TSG analyses the error report and provided correction suggestions and adjusts the existing tests to match the correct program behavior. The TSG is explicitly instructed not to delete or bypass failing tests but to align them with the PUT's actual functionality, assuming the PUT itself is fault-free.
>
> **Refinement**: Triggered by *"Refine the existing test suite for the local project* `<projectname>` *based on the latest evaluation!"*.
> This mode incorporates a high-level evaluation from the TSE of the current test suite's state. The TSG interprets this feedback as a natural language gradient describing optimization directions such as expanding coverage, testing unverified edge cases or improving assertion granularity. Each refinement iteration aims to move the suite closer toward maximal adequacy and coverage without redundancy.

Throughout all modes, the TSG enforces critical generation principles embedded in its system prompt:

> **Feedback Priority**: When a local evaluation report is present, its feedback takes precedence over the original requirements. The TSG always adapts tests to reflect the evaluated behavior of the PUT rather than the initial specification if they do not align.

Failing Test Fixes: When correcting failing tests, the TSG assumes that the program implementation is correct. Tests must therefore be adjusted to match the actual observed behavior without altering the intended logic to force a passing result. All corrections are grounded in the provided analysis rather than heuristic assumptions.

Coverage Context: The TSG focuses exclusively on meaningful, reachable coverage. It avoids creating tests for unreachable code paths and uses conditional skipping for platform-specific branches that cannot execute on the current system. Only testable and practically reachable paths are included.

Test Quality Standards: Each generated test must clearly indicate its purpose through descriptive naming and well-defined parameterization. Similar tests with input variations use `pytest.mark.parametrize` and both positive and negative test cases are included to ensure comprehensive behavioral coverage.

Error Handling: The TSG tests only realistic error conditions that can actually occur at runtime. Exception handling is validated explicitly using `pytest.raises`, ensuring correctness of error pathways without introducing artificial or unreachable conditions.

All file interactions are limited to the `tests/` directory. No modifications outside this folder are permitted, ensuring complete containment within the defined environment and most importantly no possibility of changes to the PUT to align its behavior to the existing tests. All generated or updated tests follow a predefined template and preserve traceability by linking each test to its corresponding requirement identifier. Once the new or modified suite is written, the agent signals completion and terminates, passing control to the execution phase in this iteration.

4.2.2 Test Suite Debugger

The Test Suite Debugger acts as the analytical counterpart to the TSG and performs the diagnostic phase of the repair loop. It is responsible for identifying and explaining all failures that occur during the collection or execution of the test suite. Operating under the assumption that the PUT itself is fault-free, the TSD traces each failure back to incorrect or inconsistent tests.

The TSD is invoked using the instruction: *"Error(s) during the collection or fail(s) occurred during execution of the test suite for the local project* `<projectname>`*! Please analyse them!"* This instruction triggers the debugger to analyse all failure artifacts generated during the latest execution phase. Its task is purely diagnostic as it never modifies the test suite directly but instead outputs a detailed fix report. The separation of analysis and modification ensures determinism and avoids cascading side effects.

The TSD's operation is divided into six sequential phases as defined in the system prompt (Subsection A.2):

1. **Project Discovery**: The agent explores the project folder structure, identifies all source and test files, and collects references to all failing nodes reported in the previous execution.

2. **Failure Ingestion & Grouping**: Failures are parsed from the `xml` and textual reports. Related errors are clustered by exception type, failure site, or common traceback to avoid redundant analyses.

3. **Evidence-Driven Diagnosis**: For each failure cluster, the TSD inspects relevant stack frames, captured logs, and fixtures to locate the root cause. Each finding must be grounded in concrete evidence rather than heuristic reasoning.

4. **Root-Cause Classification**: The agent classifies each issue according to five possible defect origins: (a) implementation logic, (b) test assumption or fixture, (c) configuration or import, (d) platform or reachability, and (e) environment. Since the PUT is presumed correct, only (e) provides a valid reason to skip the affected tests.

5. **Fix Proposal Generation**: For every failure cluster, the debugger proposes corrective actions in two complementary forms: a textual explanation and a minimal code patch (provided as a unified diff or snippet). Fixes must be precise, reproducible, and never suppress failures by skipping or marking tests as expected failures.

6. **Safety & Summary Reporting**: Before concluding, the debugger verifies that no fixes contradict the test independence principles or global fixture constraints. It then emits a summary table listing all analysed failure clusters, their cause categories, and the proposed next steps.

The TSD's analytical output adheres to a strict template comprising: (1) error description, (2) root-cause analysis, (3) textual fix proposal, (4) code patch, and (5) impact assessment on coverage and requirement linkage. This structure ensures traceability between observed errors, their causes, and the corrective actions implemented in the following iteration.

The TSD's workflow embraces several critical principles:

Evidence over Intuition: All conclusions must cite specific files, lines, or frames within the failure trace.

Meaningful Corrections: Proposed changes must address actual causes, not symptoms; blind skipping is prohibited.

Contextual Reachability: Unreachable or platform-gated branches must be recognized and marked with justified `skipif` guards rather than failing assertions.

Minimal, Safe Diffs: The TSD must propose the smallest code changes necessary to fully resolve a failure.

Traceability: Each fix must reference the failing test identifiers, the related requirements, and the coverage implications.

The resulting file provides the TSG with targeted guidance on which parts of the test suite must be corrected and how.

4.2.3 Test Suite Evaluator

The Test Suite Evaluator represents the evaluative component of the LIFT architecture and feedback channel within the optimization loop. While the TSD identifies incorrect behavior, the TSE determines the adequacy and effectiveness of a fully passing test suite. Its role is to analyse the suite's overall quality, identify meaningful coverage gaps, and determine whether the test suite requires further refinement. Following the TextGrad terminology, the TSE generates the natural language gradient, guiding the TSG toward areas of improvement through high-level feedback.

The TSE is invoked using the instruction: *"Evaluate the given test suite for the local project* `<projectname>` *based on the latest execution reports!"* This instruction triggers the evaluator to analyse all available artifacts from the most recent execution phase, including the statement and branch coverage report, the overall test results, and the test suite itself. The evaluation does not alter any files directly but produces a detailed report following a predefined template structure (Appendix B). This report serves as the guiding document for the next refinement iteration performed by the TSG.

The TSE's behavior is defined by a structured system prompt (Subsection A.3) describing its role, environment, and workflow. Its workflow proceeds through six distinct phases:

1. **Project Discovery**: The evaluator inspects the local project directory to locate all source files and existing tests within the `tests/` folder.

2. **Comprehensive Reading**: The evaluator reads the program source code, the functional requirements, and the generated reports to build a contextual understanding of the project behavior and test coverage.

3. **Template Review**: The evaluation template is read and prepared for population with the analysis results.

4. **Deep Analysis**: The evaluator conducts a detailed examination of the execution and coverage reports. Coverage functions as the principal quantitative indicator for this evaluation phase. Statement and branch coverage are interpreted not as absolute targets but as contextual metrics. They are numerical signals that are used alongside qualitative assessments of test adequacy, requirement fulfillment, and behavioral completeness.
   Uncovered lines and branches are identified and categorized according to their contextual relevance:

   - *Unreachable or defensive code* (e.g., error handlers for impossible conditions) is recognized as non-critical.
   - *Platform-specific branches* are considered untestable if they cannot execute on the current system.
   - *Meaningful coverage gaps* - testable but currently untested scenarios - are highlighted as genuine deficiencies requiring additional test cases.

5. **Evaluation Report Generation**: Based on the analysis, the TSE writes an evaluation report summarizing:
    - uncovered or under-tested requirements,
    - missing edge or boundary value tests,
    - untested integration scenarios, and
    - recommendations for improving coverage and test adequacy.
6. **Final Decision**: The agent concludes with a decision marker:
    - `<REWORK>` if the suite has significant testable coverage gaps, unfulfilled requirements, or needs other adjustments to achieve acceptable quality,
    - `<FINAL>` if the suite achieves sufficient contextual coverage and quality.

The TSE's evaluation process follows several critical principles embedded in its system prompt:

> **Requirement Testing**: All functional requirements must be adequately represented by corresponding test cases.
>
> **Coverage Context**: Not all uncovered code constitutes a problem - unreachable or defensive code is recognized and excluded from improvement recommendations.
>
> **Practical Focus**: Only meaningful and achievable improvements are proposed; the TSE avoids arbitrary coverage targets such as enforcing 100% coverage.
>
> **Detailed Justification**: Every recommendation is supported by explicit evidence from the code base or coverage reports and includes a clear rationale.
>
> **Actionable Feedback**: Suggestions are expressed as concrete, implementable improvements that can directly guide the next refinement iteration.

The resulting evaluation file serves as the feedback channel for LIFT. It represents the natural language equivalent of a gradient update, in which measured coverage improvements provide the primary quantitative signal. At the same time, qualitative evaluations refine the suite's structural and semantic quality and robustness.

4.3 Metrics & Traceability

In pursuit of full-scope automated test development, LIFT implements an explicit traceability framework and systematic metric recording across all iterations. Reflecting the principles of the Waterfall and V-Model processes (Section 2.2), traceability forms the backbone of Verification & Validation. It establishes bidirectional links between requirements, generated specifications, and the resulting tests, thereby ensuring that every behavioral requirement can be traced to one or more verifying tests and that each test can be traced back to its motivating requirement. Such explicit linkage is essential not only for quantitative assessment but also for human trust and auditability of automatically generated suites.

Figure 14: Relation between requirements, test specifications, and tests within LIFT

As shown in Figure 14, the requirement's specification is defined as an external LIFT input in the `program-requirements.yml`. For each individual test, a natural language specification is added during test creation by the TSG that describes its intent and references the corresponding requirement identifiers. These links are encoded in the test metadata fields used by the reporting system. This representation allows for both one-to-many and many-to-one relations: a single requirement may be covered by multiple tests and a single test may contribute to validating multiple requirements.

Quantitative metrics are collected after every execution phase of an iteration, providing a numerical record of the test suite's current state. The recorded metrics include:

- **Reliability metrics**: number of errors; counts of passed, failed, and skipped tests; and total number of executed tests,
- **Performance metrics**: total execution time, and
- **Coverage metrics**: statement and branch coverage, which serve as the primary quantitative indicators of test adequacy.

Among these, coverage constitutes the main quantifiable signal guiding the Test Suite Evaluator's feedback process, while the remaining metrics are also available to the TSE and provide contextual insight into suite stability, completeness, and efficiency. For each iteration, the complete set of metrics is appended to the project archive, allowing temporal comparison of results across successive refinements.

The evolution of these metrics over time forms the *iterative dynamics* of the system. These trajectories describe how quantitative indicators change as the suite undergoes generation, repair, and evaluation cycles. They offer a measurable reflection of the optimization progress: while the TSE provides qualitative gradient feedback, metric trajectories expose its quantitative counterpart, capturing the observable improvement of the test suite across iterations.

LIFT implements multiple options to determine whether the current test suite's state is sufficient.

1. **Metric Thresholds**: Thresholds for numeric metrics like mutation score (if available), coverage, or test suite size can be defined.
2. **TSE refinement decision marker**: The TSE's feedback includes a decision marker that summarizes whether the TSE deems the suggested enhancements to be necessary in order to achieve a sufficient test suite.

By default, the iterative refinement process terminates once the TSE's feedback indicates that no significant improvement is required. During execution, all mentioned metrics are continuously generated and archived, forming a robust basis for alternative termination algorithms or enforced threshold-based stopping criteria.[5]

[5]Note that the number of iterations also provides a termination criterion. Since it's not related to the test suite's sufficiency, it is not mentioned here.

5 Case Study: `simplejson`

To demonstrate and evaluate the applicability of the proposed LIFT framework, a representative open-source Python project was selected as the Program under Test. This section introduces the library `simplejson`, outlines its functional purpose, structure, and characteristics and discusses relevant aspects influencing its testability. The objective is to provide sufficient contextual understanding of the Program under Test prior to presenting the evaluation results in Section 6.

`simplejson` is a lightweight, high-performance library for encoding and decoding data in the JavaScript Object Notation (JSON) format. It is developed and maintained as an independent project by Bob Ippolito and has been continuously updated since its initial release in 2006 [20]. The library is independent of Python's built-in `json` module, which is now part of the Python Standard Library. `simplejson` remains in active use due to its extended feature set, configurability, and continued compatibility with both Python 2 and Python 3.

The library was chosen as the Program under Test for several reasons:

- It represents a **pure Python project** with moderate complexity, featuring multiple modules that interact in well-defined ways.
- Its functionality - translating between Python data structures and serialized JSON text - is highly deterministic, making it suitable for automated and reproducible testing.
- The codebase combines both algorithmic logic (e.g., parsing, type handling) and configuration-driven behavior, allowing diverse test generation scenarios.
- As an open-source project widely adopted across the Python ecosystem, it serves as a realistic and generalizable benchmark for evaluating LLM-based test suite generation.

5.1 Library Functionality

The main purpose of `simplejson` is to provide a complete implementation of JSON serialization and deserialization for Python objects. Its core components are:

- `JSONEncoder`: responsible for transforming Python objects into valid JSON strings. It supports basic data types such as integers, floats, strings, lists, and dictionaries as well as complex structures including nested or custom objects through hooks like `default()` and `for_json()`.
- `JSONDecoder`: performs the inverse operation, reconstructing Python objects from JSON text representations. It handles input validation, type inference, and optional strict parsing modes.
- **Utility Functions**: the high-level API functions `dump()`, `dumps()`, `load()`, and `loads()` provide convenient interfaces for reading and writing JSON data to and from strings, files or streams.

The implementation adheres to the RFC 8259 standard for JSON [6] while extending it with optional features for greater flexibility. These include support for `Decimal` objects, custom `namedtuple` encoding, and control over key ordering in dictionaries.

Beyond its functional completeness, `simplejson` offers additional methodological advantages for evaluating LLM-based test generation. As JSON is a foundational data-interchange format across virtually all programming languages, its semantics and usage patterns are deeply embedded in the training corpora of Large Language Models. This property potentially mitigates the oracle problem, since the model can accurately predict expected input–output behavior without rigorous external specifications. Moreover, the library's extensive inline documentation and descriptive docstrings, which form the basis of its public documentation, provide rich contextual cues to guide reasoning during generation (see Listing 1). Finally, the library's interaction with various I/O channels - such as files, streams, and terminal interfaces - adds an additional layer of operational complexity that goes beyond algorithmic or mathematical test subjects, making it a more representative benchmark for practical Software Testing.

```
r"""Command-line tool to validate and pretty-print JSON

Usage::

    $ echo '{"json":"obj"}' | python -m simplejson.tool
    {
        "json": "obj"
    }
    $ echo '{ 1.2:3.4}' | python -m simplejson.tool
    Expecting property name: line 1 column 2 (char 2)

"""
```

Listing 1: Docstring of the `tool.py`

This example illustrates the clarity and task-oriented phrasing found throughout `simplejson`'s source files. Such descriptive docstrings not only aid human maintainers but also provide contextual signals that can improve an LLM's reasoning during automated test generation.

5.2 Structural & Functional Overview

The project follows a modular structure typical for medium-sized Python libraries. Its main package `simplejson` contains several source files, each implementing specific aspects of the encoding and decoding process. `simplejson`'s package layout is shown below:

```
simplejson/
├── __init__.py      (23KB,  82 exec. lines, 562 total lines of code)
├── _speedups.c      (excluded)
├── compat.py        ( 1KB,  29 exec. lines,  34 total lines of code)
├── decoder.py       (15KB, 229 exec. lines, 416 total lines of code)
├── encoder.py       (29KB, 412 exec. lines, 740 total lines of code)
├── errors.py        ( 2KB,  29 exec. lines,  53 total lines of code)
├── ordered_dict.py  ( 3KB,  81 exec. lines, 103 total lines of code)
├── raw_json.py      ( 1KB,   3 exec. lines,   9 total lines of code)
├── scanner.py       ( 3KB,  64 exec. lines,  85 total lines of code)
└── tool.py          ( 2KB,  24 exec. lines,  42 total lines of code)
```

The modules `encoder.py`, `decoder.py`, and `scanner.py` form the functional core of the library. The file `ordered_dict.py` provides a backport of `OrderedDict` for Python 2 environments, while `compat.py` ensures compatibility across interpreter versions. The optional C extension `_speedups.c` implements performance-critical parts of the encoder and decoder to accelerate runtime execution. It will be excluded from testing due to limitations of `pytest`.

In total, the codebase comprises a total of 953 executable lines of code and 462 branches distributed across the 9 Python source files. The relative size and cohesion of these modules make `simplejson` a suitable target for evaluating both functional and structural coverage metrics.

At runtime, the encoder and decoder operate as separate but complementary components. The encoding process begins with user-facing calls to `dump()` or `dumps()`, which internally instantiate a `JSONEncoder` object configured according to user parameters such as indentation, key sorting or special value handling (e.g., `NaN` or `Infinity`). The encoder recursively traverses the Python object graph and converts each element into a valid JSON fragment. Decoding follows the opposite direction. The `load()` and `loads()` functions delegate to the `JSONDecoder`, which uses a lexical scanner implemented in `scanner.py` to tokenize the input and rebuild the corresponding Python structures. Error handling for malformed JSON or unsupported data types is implemented through well-defined exceptions such as the `JSONDecodeError`.

5.3 Comparison with the Python Standard Library

Although the Python Standard Library includes a built-in `json` module, `simplejson` remains useful due to its additional configuration options. The two libraries share a common interface design. The major differences include:

- **Extended feature set**: options such as `use_decimal`, `namedtuple_as_object` and `for_json` provide greater customization for complex objects.
- **Performance optimizations**: the optional `_speedups` C extension significantly reduces parsing and encoding times for large datasets.

- **Backward compatibility**: explicit support for legacy constructs like `OrderedDict` and Python 2 string types.
- **Independent maintenance**: separate release cycle, enabling faster adoption of JSON standard revisions or bug fixes.

From a testing perspective, these differences introduce unique challenges. Configuration-dependent behaviors lead to a combinatorial explosion of possible test inputs and differences between Python versions affect reproducibility and expected outputs.

5.4 Testability Considerations

`simplejson` is well-suited for evaluating automated test generation approaches due to its deterministic behavior and well-defined function boundaries. Nonetheless, certain parts of the codebase are less suitable to LLM-based test generation:

- **Cross-version compatibility**: Python 3 functionality implemented using Python 2 source code (i.e., `OrderedDict`) is not testable in modern Python 3 environments.
- **Native extensions**: the C-based module `_speedups.c` cannot be directly tested using Python-based frameworks and must therefore be excluded.
- **Platform dependencies**: minor variations in floating-point precision or Unicode handling can produce environment-specific outputs.

Despite these limitations, the majority of the codebase remains accessible for automated testing. The functional purity of the encoder and decoder allows for repeatable test execution and deterministic comparison of expected versus actual outputs. Additionally, the library's comprehensive use of exception handling provides rich oracles for assessing behavioral correctness.

6 Evaluation

The evaluation aims to assess the effectiveness and quality of the proposed LIFT framework. It builds upon established metrics and methodologies from prior research in LLM-based test generation, combining both quantitative and qualitative dimensions of test suite quality. The following sections first summarize these evaluation criteria adopted from Related Work (Section 3) to create a performance baseline before applying them to the conducted case study on `simplejson`.

6.1 Evaluation criteria from related research

ChatTester Yuan et al. present ChatTester [38], one of the earliest systematic evaluations of LLM-based unit test generation. Building upon prior work such as ChatUniTest [9], they extend the evaluation to include both quantitative and qualitative aspects of generated test quality. In [38], they describe four fundamental questions:

"How is the correctness of the unit tests generated by ChatGPT?" In order for any test generation method to be useful, the generated tests need to be correct. *Correctness* itself has multiple facets. Firstly, code must be syntactically correct, must compile, and then must execute correctly. A correct execution itself consists of (a) no runtime errors and (b) no incorrect (failing) assertions. A breakdown of occurring errors indicates which complexity layers can be solved by LLMs consistently and which represent a boundary for current LLM abilities. [38]

"How is the sufficiency of the unit tests generated by ChatGPT?" The correctness of generated tests alone does not reflect their ability to test a PUT in a qualitative way. *Sufficiency* attempts to indicate the quality of the generated test through proxy metrics like coverage and number of assertions in the test case[6]. [38]

"How is the readability of the unit tests generated by ChatGPT?" *Readability* is a metric inherent to automated test case generation. Yuan et al. rely on a user study to evaluate this subjective metric. Beyond the quantifiable measures, this metric indicates how human-like the generated test cases are. This subjective evaluation captures human-perceived qualities such as meaningful naming, structural clarity, and code formatting, complementing the objective metrics above. [38]

"How can the unit tests generated by ChatGPT be used by developers?" Yuan et al. define ChatTester [38] including one of the first systematic evaluations of LLM-based unit test generation. In their study, only 73.3% of generated tests are compilable and just 41.0% of all generated tests pass the execution, showing that human intervention remained essential for achieving production-ready quality. [38]

TestART TestART [15] represents the most recent and comprehensive work on LLM-based test generation and is conceptually closest to the proposed LIFT, as it also generates and iteratively refines whole test suites at the PUT level, achieving a 78.55% pass rate and a line coverage of

[6]Since ChatTester [38] only creates unit tests for a single focal method as its input, the number of assertions is measured for the generated test case.

just 68.17%. In their evaluation, Gu et al. extend prior correctness and sufficiency metrics with mutation-based fault-detection measures. They measure TestART's performance by the following key perspectives:

1. **Correctness**
    - *Syntax Error*: percentage of test code that includes Java syntax errors,
    - *Compile Error*: percentage of test code that produces errors during the compilation,
    - *Runtime Error*: percentage of test code that includes errors or failures during the execution, and
    - *Pass Rate*: percentage of test code that runs without errors, failures, failed assertions, or other problems.
2. **Sufficiency**
    - *Branch coverage of correct tests*: branch coverage rate of the passed focal methods,
    - *Line coverage of correct tests*: line coverage rate of the passed focal methods,
    - *Total branch coverage*: branch coverage over all focal methods and
    - *Total line coverage*: line coverage over all focal methods.
3. **Error detection**
    - *Mutation coverage*: number of killed mutants over the total number of created mutants and
    - *Test strength*: number of killed mutants over the number of mutants covered by tests.
4. **Test case count**
    - *Test case count*: total number of tests generated by the testing approach and
    - *Assertion count*: total number of assertions within the created tests. [15]

MuTAP Dakhel et al. introduce MuTAP [10] as a mutation-testing–driven refinement framework for LLM-generated unit tests. Building upon the generation-validation-repair paradigm of ChatTester, MuTAP evaluates the effectiveness of its iterative repair mechanism using a set of mutation-based and validity metrics. Specifically, the authors measure
(a) the *validity rate* - the percentage of syntactically correct, compilable, and executable tests,
(b) the *pass rate* of tests that execute successfully and
(c) the *mutation score* (*MS*), defined as the ratio of killed mutants to all generated mutants. [10]

Across repair iterations, the improvement in mutation score (ΔMS) is used to quantify how LLM-guided refinements enhance fault-detection capability. Dakhel et al. explicitly note that "improving MS does not necessarily lead to better coverage and that test coverage is only weakly correlated with the efficiency of tests" [10, p. 13], emphasizing that mutation-based evaluation captures a complementary aspect of test quality. [10]

***RQ-2.3*: What constitutes a good test suite in the context of LLM-based generation?**

Research into LLM-based test case generation shows that similar metrics can be applied to this generation method as well. Both dimensions of test suite quality - quantitative and qualitative - need to be fulfilled and analysed.
The generated tests must be syntactically correct, compilable, and executable without requiring human intervention. High coverage and mutation scores can indicate that test suites are effective. Equally important is *semantic adequacy* - the alignment between the intent of the test and the intended behavior of the PUT - which ensures that generated assertions meaningfully capture expected functionality rather than coincidental output patterns.
From a qualitative perspective, a good LLM-generated suite should remain *maintainable*, *readable*, and *trustworthy*, minimizing redundancy and ambiguity. Its usefulness is further determined by *developer adoptability*: tests must be interpretable and logically structured so that human developers can review, adapt, and extend them with minimal friction.
In summary, a good test suite in the LLM-generation context unifies quantitative soundness - valid, executable, and fault-revealing tests - with qualitative robustness - readable, reliable, and semantically meaningful tests that can be trusted and maintained in long-term development. [10, 15, 38]

6.2 Evaluation of the Case Study

The evaluation of LIFT aims to determine its effectiveness in generating complete and executable test suites that improve in quality through iterative refinement. Following the quantitative and qualitative criteria defined in Subsection 2.3 and Subsection 6.1, this section assesses the generated suites with respect to test suite size, correctness, coverage, mutation score, and semantic quality.

The open-source library `simplejson` [20] serves as the PUT. It provides a realistic and well-documented target featuring complex serialization logic, error handling, and a Command-Line Interface. To ensure comparability and reproducibility across trials, all experiments were conducted within the following environment:

- Python 3.12.10
- `pytest` (8.3.5) and `pytest-html-report` (1.0.6)
- `pytest-cov` (6.1.1)
- `mutmut` (2.5.1) and `parso` (0.8.4)

The LIFT framework was executed with AI-generated requirements verified by humans prior to the experiments. In total, 39 requirements were defined, covering encoding and decoding behavior, error handling, and CLI interactions. An excerpt of the requirements document (Appendix C) is shown in Listing 2.

```
simplejson_test_requirements:
  scope_and_environment:
    - id: SCOPE-1
      title: Public API exposure
      description: The package must provide the public functions and classes
       ↪ defined in its documentation: dump, dumps, load, loads, JSONEncoder,
       ↪ and JSONDecoder. This ensures compatibility with the Python stdlib
       ↪ json API.
      acceptance: Importing simplejson exposes the documented functions and
       ↪ classes.
  ...

  encoding:
    core_correctness:
      - id: ENC-CORE-1
        title: dumps returns JSON string
        description: Calling dumps(obj) must return a Python str containing
         ↪ valid JSON that conforms to RFC rules. The returned type must never
         ↪ be bytes.
        acceptance: Type is str; round-tripping via loads yields equivalent
         ↪ Python objects.
```

Listing 2: Excerpt from the requirement specification document

A total of 24 independent trials were executed, each using `gpt-5-mini` with a fixed temperature of 1.0[7] as the underlying model for all LLM agents. Every trial was run for 25 iterations without early termination, resulting in 24 refinement iterations following the initial generation.

Two types of resulting test suites were distinguished:

- **First Sufficient Test Suite (FSS)**: The first suite in each trial deemed sufficient by the TSE according to its own evaluation criteria. Some trials may not reach a sufficient suite within the iteration limit and therefore have no FSS.
- **Last Passing Test Suite (LPS)**: The final suite that executed without any errors or assertion failures, regardless of TSE evaluation.

The original test suite included with `simplejson` was not included during the trial generations but serves as a human-crafted reference for comparative analysis. It represents the target quality level that LIFT-generated suites are expected to approach in terms of coverage and robustness.

To assess the completeness of the generated test suites, mutation testing was conducted for all of them. The same set of mutants was applied to the original suite as well as to each trial's FSS and LPS, and the resulting mutation score (*MS*) was computed. It is important to note that mutation testing is not part of the LIFT process and is therefore not explicitly optimized by the Test Suite Evaluator. The *MS* serves as an external proxy metric for the general robustness and fault-detection capability of the generated suites, as no reward shaping or "reward hacking" can occur against this unknown objective. Consequently, a significantly high *MS* can be interpreted as evidence that a test suite closely envelops the behavioral space of the PUT.

6.2.1 General and Test Counts

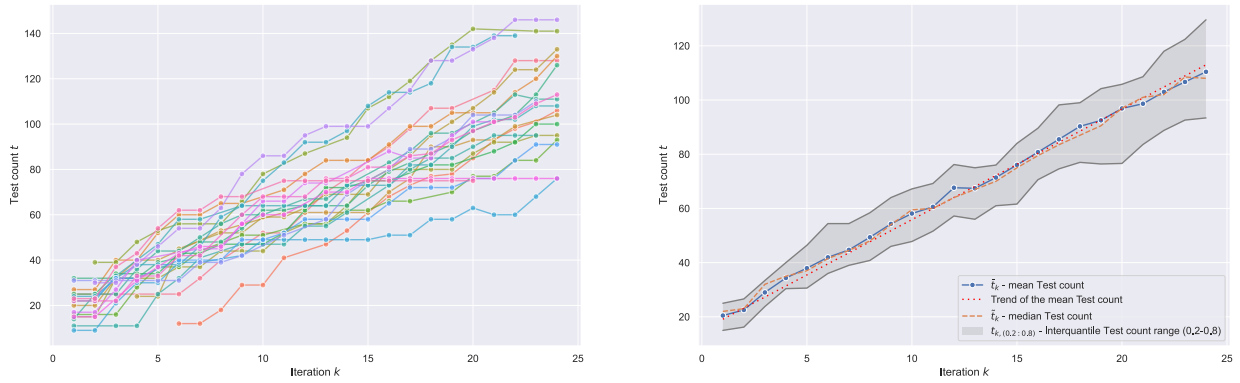
LIFT successfully produces complete test suites and consistently extends them over the course of its iterations. As shown in Figure 15, the number of tests steadily increases across all trials without showing signs of saturation within the fixed 25 iterations. This confirms that the underlying LLM is not only capable of generating an initial executable suite but also of adding further tests to an existing one. Whether this continuous growth reflects a meaningful extension of behavioral coverage or merely expansion without necessity cannot be inferred from test suite size alone.

The observed growth pattern suggests a general preference of the LIFT agents to create new test cases rather than adapt or refine existing ones. This behavior may stem from an implicit bias to avoid modifying already functional test code, thereby reducing the risk of introducing new errors into previously passing parts of the suite.

Two of the 24 trials (IDs 14 and 15) exhibited a complete breakdown of their test suites, where the TSG removed most tests after an earlier period of stable growth. These trials were excluded from the aggregated statistics in the following analysis to preserve comparability, as such suites would rightfully be rejected by any human developer in practice. 12 of the 24 trials reached a test suite that was evaluated as sufficient by the Test Suite Evaluator.

[7] Note that during time of execution, it was not possible to change the temperature for the `gpt-5-mini` model via the `OpenAI` API.

Figure 15a illustrates the evolution of total test counts for each trial. All initial test suites start with roughly 10 to 30 test cases, but none of them executed successfully initially due to import-related issues, indicating that tests were not adequately adapted to their environment. The aggregated statistics shown in Figure 15b (mean, median, and quantiles) only include iterations where no errors were reported, since test count reports are distorted by `pytest` when suites fail during collection or execution. The results demonstrate a consistent linear growth of the test suites, averaging approximately 4.1 newly added tests per iteration.



(a) Individual test counts per trial
(for all test suite iterations without errors)

(b) Aggregated test counts over all trials
(for all test suite iterations without errors)

Figure 15: Test counts for all trials

The original `simplejson` test suite contains a total of 143 tests, with seven skipped due to the missing import of the C speedup extension. In comparison, the average FSS consists of 64 tests (ranging from 27 to 118), while the average LPS comprises 107 tests (ranging from 68 to 146).

Trial ID	**Total Tests**	**Skipped Tests**	**Execution Times (*s*)**	**Unit Tests**	**Integration Tests**	**System Tests**
original	143	7	6.397			
FSS (mean)	64	1	3.198	60	4	0
FSS (median)	59	1	2.505	57	4	0
LPS (mean)	107	1	5.681	100	7	1
LPS (median)	106	0	5.480	102	7	0

Table 1: Composition of the original Test Suite and the FSSs & LPSs of all trials (aggr.)[8]

The sizes and execution times reported in Table 1 indicate that later iterations produce suites whose characteristics increasingly resemble those of the original `simplejson` suite. It remains

[8]Note that tests can be assigned multiple types by the LLM. Therefore, the combined number of unit, integration, and system tests must not equal the total number of tests.
Find the non-aggregated data in Subsection D.1.

unclear whether this represents a natural convergence in test suite size or is merely coincidental given the fixed limit of 25 iterations. In terms of test type distribution, the majority of generated tests are labeled as unit tests, with only a small subset classified as integration or system tests. This is consistent with the contained nature of the PUT whose functionality primarily revolves around serialization and deserialization logic contained within isolated functions. However, it could indicate an inherent limitation of LLMs in reasoning across multiple abstraction levels or distinguishing between testing scopes. Since the chosen PUT does not include complex external interactions or components that require system level testing (such as a GUI), the origin of this observed distribution cannot be conclusively attributed to either factor based on the amount alone.

Overall, LIFT demonstrates stable and continuous test suite growth, without conclusive evidence of convergence toward a size comparable to the human-written baseline. The results confirm its capacity for iterative expansion while raising open questions about whether this growth primarily expands structural completeness or increases behavioral redundancy.

6.2.2 Correctness

Unlike other frameworks such as TestART [15] or ChatUniTest [9], LIFT only outputs fully passing test suites. Non-executable or failing intermediate versions are therefore not part of its externally visible output.

Across all trials, the first fully passing test suite was typically obtained after two refinement iterations following the initial generation. This behavior was consistent, as the initial test suites failed due to missing imports of the `simplejson` package, which was not preinstalled and hence unavailable when referenced directly. The TSD persistently suggested the correct fixes to resolve this issue during its first intervention by adapting the import handling, after which the suites became executable. Once these environment-related problems were corrected, subsequent failures were largely confined to assertion-level mismatches, which were then addressed in the following refinement step.

To quantify the share of iterations devoted to fixing errors versus those focusing on refinement, two rates were defined. Let $x_k^{(j)}$ denote a binary indicator for trial j and iteration k, which equals 1 if the iteration triggered the TSD (i.e., a repair analysis) and 0 otherwise. The mean per-iteration Fix Rate $\bar{r}_k$ and the mean cumulative Fix Rate $\bar{R}_k$ are then given by:

$$\bar{r}_k = \frac{1}{m}\sum_{j=1}^{m} r_k^{(j)} = \frac{1}{m}\sum_{j=1}^{m} x_k^{(j)} \tag{4}$$

$$\bar{R}_k = \frac{1}{m}\sum_{j=1}^{m} R_k^{(j)} = \frac{1}{m}\sum_{j=1}^{m}\frac{1}{k}\sum_{i=1}^{k} x_i^{(j)} \tag{5}$$

where m denotes the total number of trials. $\bar{r}_k$ thus represents the average proportion of trials invoking the TSD in iteration k, while $\bar{R}_k$ expresses the mean cumulative share of refinement iterations up to and including iteration k that triggered a TSD investigation.

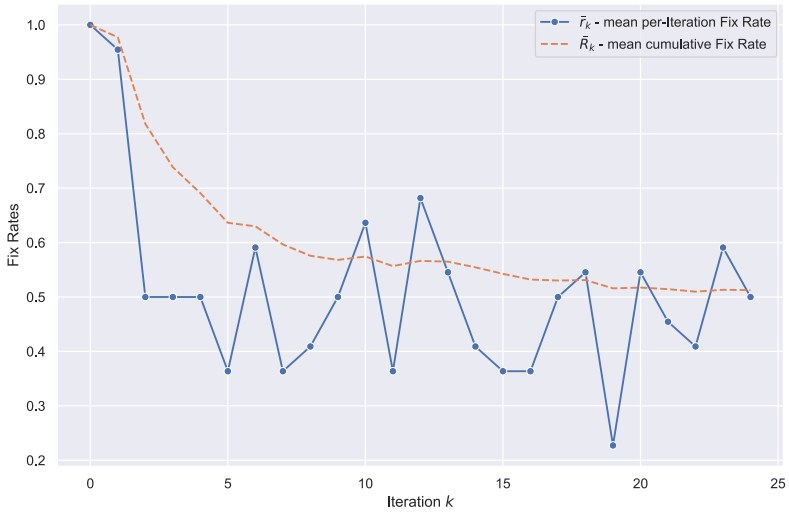


Figure 16: Aggregated Fix Rates for all trials

As expected, both ratios ($\bar{r}_k$ and $\bar{R}_k$) start high, reflecting the initial adjustment phase in which the generated tests must first be aligned with the execution environment. Over time, the cumulative ratio $\bar{R}_k$ stabilizes at approximately 0.5, indicating that roughly half of the iterations include test suites that encounter errors or assertion failures, while the other half are stable and receive feedback from the TSE. The solid line in Figure 16 represents the per-iteration rate $\bar{r}_k$, which fluctuates around the same level but exhibits considerable variance between individual iterations.

This distribution is notable, as one might intuitively expect a larger share of iterations to be devoted to bug fixing rather than improvement and expansion. The roughly balanced ratio suggests that either (a) most technical issues are resolved quickly - typically within one or two iterations - or (b) the refinement process itself is stable enough to avoid introducing new faults. In either case, the data indicate that LIFT's feedback loop maintains a steady balance between correctness (ensured by the TSD) and growth (driven by the TSE).

Correctness in LIFT is treated as a baseline condition rather than an optimization objective. The generated test suites remain consistently executable across iterations, once initial test environment based issues are resolved. This design choice differentiates LIFT from other LLM-based test generation approaches, which frequently output partially failing or non-compiling test sets. The effectiveness of LIFT's repair mechanism ensures that its refinement cycles operate exclusively on valid and fully executable test suites throughout the process.

6.2.3 Structural Sufficieny and Coverage

The initial structure of the generated test suites is logically organized and follows the instructed one-to-one mapping between source files and their corresponding test files. During the fixing iterations, this structure remains stable, as only existing tests are modified. However, in refinement iterations - aiming to extend the suites toward sufficiency as requested by the Test Suite Evaluator - the directory structure consistently deteriorates. This manifests in an uncontrolled creation of

numerous new test files intended to close behavioral gaps identified by the TSE with non-descriptive names like:

- `test_errors_additional.py`,
- `test_evaluation_refinements_additional.py`,
- `test_evaluation_refinements2.py`,
- `test_refinements_additional.py` or
- `test_refinements_evaluation2.py`.

The result is a fragmented and untraceable suite organization, suggesting a limitation of the underlying LLM in maintaining an internal representation of the existing test hierarchy and integrating new tests into coherent clusters.

Beyond structural organization, the provided coverage of a test suite is important in analysing its sufficiency. Figure 17a shows that even the first passing test suites[9] already achieve an average line coverage exceeding 60%. Coverage increases steadily across iterations, with the average LPS reaching over 83% line coverage. The average FSS attains roughly 75% coverage (around 7 percentage points lower than the original suite), while the LPS exceeds the baseline by about 1–2 percentage points (Table 2). This demonstrates that LIFT can, in principle, reach or slightly surpass the structural adequacy of the manually written test suite - even with significantly lower test counts[10].

Trial ID	**Total Lines**	**Covered Lines**	**Line Coverage (%)**	**Total Branches**	**Covered Branches**	**Branch Coverage (%)**
original	953	786	82.48	462	396	85.71
FSS (mean)	953	720	75.60	462	317	68.75
FSS (median)	953	704	73.87	462	319	69.05
LPS (mean)	953	798	83.84	462	350	75.91
LPS (median)	953	809	84.89	462	351	75.97

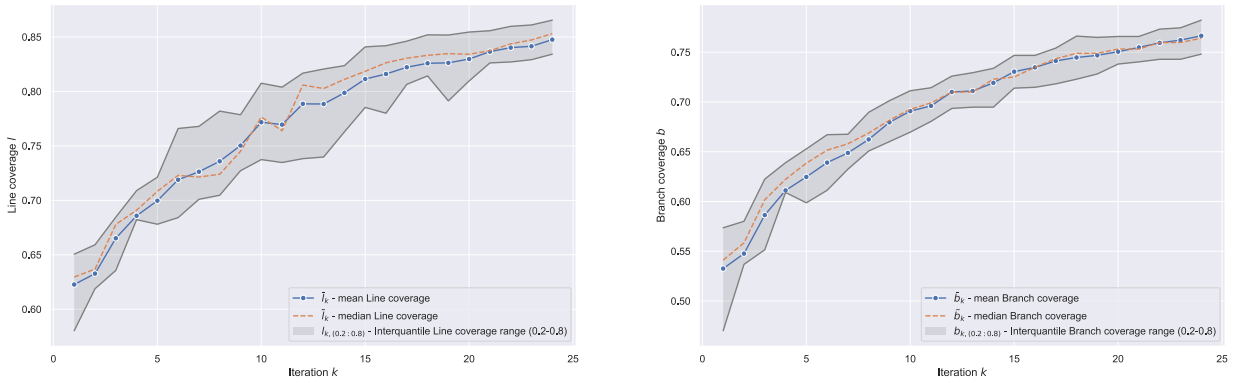
Table 2: Coverages of the original Test Suite and the FSSs & LPSs of all trials (aggr.)[11]

Branch coverage exhibits a similar but less pronounced growth trend (Figure 17b). While initial suites begin with comparable values to line coverage, their improvement is slower, resulting in an average of only 76% for the LPS. This is approximately 7 percentage points above the corresponding FSS value but still 10 points below that of the original suite (Table 2). This suggests that the generated tests tend to exercise many code paths superficially but fail to capture deeper branching behavior indicative of more complex logical variations.

[9] Commonly, this is the initial test suite with the applied fixes to integrate into the test environment and correct failing assertions. The first passing test suite is not equal to the First Sufficient Test Suite!

[10] The average LPS is around 75% the size of the original test suite.

[11] Find the non-aggregated data in Subsection D.2.



(a) Line Coverage for fully passing test suites

(b) Line Coverage (aggr.)

Figure 17: Aggreated coverages for all trials

Taken together, the coverage results reveal that LIFT-generated suites achieve structural sufficiency in terms of broadness but not necessarily in behavioral depth. The optimization of coverage as the primary numeric feedback signal may have introduced a form of reward bias into the Test Suite Evaluator's feedback, where the TSG increases line coverage by targeting simpler statements rather than executing critical control-flow paths. This highlights a possible limitation of basing the optimization partially on this purely quantitative measure.

6.2.4 Error Detection Capability

To evaluate the fault-revealing capability of the generated test suites, mutation testing was applied as an external and independent measure. As outlined in Subsection 6.2, it is not part of the LIFT process itself and thus serves as an unbiased indicator of behavioral adequacy. A total of 1,184 mutants were generated for `simplejson` and executed against the original test suite, the First Sufficient Test Suites and Last Passing Test Suites.

Trial ID	**Total Mutants**	**Killed Mutants**	**Mutation Score** MS **(%)**
original	1184	863	72.89
FSS (mean)	1184	509	43.00
FSS (median)	1184	554	46.79
LPS (mean)	1184	703	59.36
LPS (median)	1184	709	59.84

Table 3: Mutation testing results of the original Test Suite and the FSSs & LPSs of all trials (aggr.)[12]

[12]Find the non-aggregated data in Subsection D.3.

Of the 1,184 generated mutants, 863 were killed by the original test suite, 509 by the average FSS, and 703 by the average LPS. While the refinement process within LIFT improves the mutation score from 43% (FSS) to nearly 60% in late iterations, the generated suites still fall short of the human-developed baseline by around 13 percentage points.

The increase from FSS to LPS can primarily be attributed to the larger number of tests and the correspondingly higher coverage observed in Subsection 6.2.3. As more code is executed, a greater proportion of mutants are exposed and correctly killed, demonstrating that the additional tests are at least partially effective in validating expected behavior. However, the persistent gap between the higher line coverage and the lower mutation score of the average LPS compared to the original suite indicates that the generated tests, although executing more code, do not necessarily test the behavior described within it through strict assertions. This finding is consistent with the previously observed disparity between line and branch coverage, suggesting that LIFT's optimization focuses on exercising code broadly rather than verifying complex or branching behaviors in depth.

Overall, the mutation testing results reinforce the interpretation that the LIFT-generated test suites achieve measurable improvements in structural completeness and partial fault detection but remain quantifiably inferior to manually designed tests in revealing subtle behavioral faults.

6.2.5 Holistic Coverage Exploration

All LPSs generated across the 24 trials were executed jointly as a single aggregated test suite to assess the overall behavioral coverage that the underlying LLM was capable of achieving, given sufficient exploration time, and to reveal the variance across all trials. The merged suite reached a line coverage of 96.54% and a branch coverage of 92.42%, substantially exceeding both the individual trial results and the human-written baseline. Its combined mutation score of 79.98% indicates an increased coverage of important branches, yet the growth in line and branch coverage does not translate nearly as well to a higher mutation score.

Trial ID	**Covered Lines**	**Line Coverage (%)**	**Covered Branches**	**Branch Coverage (%)**	**Killed Mutants**	**Mutation Score *MS* (%)**
original	786	82.48	396	85.71	863	72.89
FSS (mean)	720	75.60	317	68.75	509	43.00
LPS (mean)	798	83.84	350	75.91	703	59.36
all LPSs (comb.)	920	96.54	427	92.42	947	79.98

Table 4: Coverages and mutation scores of the original Test Suite, the FSSs & LPSs, and all LPSs combined

This finding highlights the latent creative capacity of the LLM to eventually exercise nearly the entire behavioral space of the PUT. It can thus be interpreted as evidence that, given sufficient computational resources and refinement iterations, LLM-driven test generation could approach exhaustive structural adequacy. In turn, this suggests that the 25 iterations employed in this study

have not yet encountered the upper limit of achievable coverage and that further improvement may remain possible through extended refinement or ensemble aggregation of independently evolved suites.

6.2.6 Behavioral Adequacy and Qualitative Test Quality

Beyond structural and quantitative aspects, the quality of the generated test suites must also be evaluated in terms of their behavioral adequacy and maintainability. In particular, this concerns the correctness of requirement linkings, the occurrence of hallucinated requirements, the overall human-likeness, and the organisation of the generated code.

Table 5 summarizes the number of distinct requirements mentioned across all trials. On average, each FSS contains 42 unique requirement identifiers, of which about 15% do not exist in the provided requirement specification. In the LPSs, the number of unique mentioned requirements increases to 48, but the share of non-existing identifiers also rises to roughly 24%. It is important to note that this value represents the percentage of *unique* identifiers that cannot be traced back to the specification, not the frequency of their occurrence. In other words, if a requirement identifier is referenced in any test of a LPS, there is an approximately 24% chance that it does not exist in the specification.

Trial ID	**Number of referenced Requirements**	**Number of referenced & existing Requirements**	**Number of hallucinated Requirements**	**Percentage of hallucinated Requirements**
FSS (mean)	42	35	7	14.81
FSS (median)	43	35	7	16.28
LPS (mean)	48	35	13	23.70
LPS (median)	48	36	13	26.53

Table 5: Requirement counts of the FSSs & LPSs of all trials (aggr.)[13]

The growing number of mentioned requirements across iterations indicates that LIFT continuously attempts to close behavioral gaps identified in the TSE evaluation based on insufficient coverage. However, a significant portion of these new tests cannot be mapped to any defined requirement. This suggests that the refinement process increasingly explores behavioral edge cases or implicit aspects of the program that are not explicitly documented. Consequently, the rise in hallucinated identifiers may point either to an insufficient coverage by the provided specification or to a limitation in LIFT's requirement-grounding mechanism, which expects a defined link for every newly introduced test.

To illustrate the correctness of requirement linkings, two representative examples were analysed: `SCOPE-1` and `ENC-CORE-1` (Listing 2). The former defines that the public API symbols of `simplejson` must be exposed upon import, while the latter specifies that `dumps()` must return a `str` and round-trip correctly through `loads()`.

[13]Find the non-aggregated data in Subsection D.4.

For SCOPE-1, nearly all FSSs included at least one correctly linked test, with only a single trial failing to cover it properly. Across all trials, 14 tests were generated for this requirement, three of which were mislinked, often referring to unrelated behavior such as the OrderedDict implementation.

```
@pytest.mark.reporting(
    developer="automatic",
    functional_specification="SCOPE-1",
    test_description="Ensure public API symbols are exposed on import"
)
@pytest.mark.category("unit")
def test_public_api_exposure():
    # Public API should expose the core functions and classes
    for name in ('dump', 'dumps', 'load', 'loads', 'JSONEncoder',
     ↪  'JSONDecoder'):
        assert hasattr(json, name), f"simplejson missing public symbol: {name}"
```

Listing 3: Example for a test covering requirement SCOPE-1

```
@pytest.mark.reporting(
    developer="automatic",
    functional_specification=["SCOPE-1"],
    test_description="Monkeypatched encode_basestring_ascii implementation is
     ↪  invoked for top-level strings"
)
@pytest.mark.category("unit")
def test_monkeypatched_encode_basestring_ascii_is_used(monkeypatch):
    called = {"flag": False}

    def fake_encode(s):
        called["flag"] = True
        return '"FAKE"'

    monkeypatch.setattr(simplejson.encoder, 'encode_basestring_ascii',
     ↪  fake_encode)
    enc = JSONEncoder()
    out = enc.encode('abc')
    assert called["flag"], "Expected the fake encode_basestring_ascii to be
     ↪  invoked"
    assert out == '"FAKE"'
```

Listing 4: Example for a test not covering requirement SCOPE-1

In the LPSs, a total of 40 tests referenced SCOPE-1, of which 22 (55%) correctly validated the intended behavior and 18 referred to other, only loosely related functionalities. All Last Passing Test Suites contained at least one correctly covering test, except for a single trial where all linkings to SCOPE-1 were incorrect.

Listing 3 shows a representative example that can be found in near identical form throughout all analysed test suites. Listing 4 highlights how the completely different behavior was falsely linked to SCOPE-1.

The second example, ENC-CORE-1, is more broadly defined and therefore seems more prone to misuse. It specifies that `json.dumps()` must return a valid JSON string and that the round-trip through `json.loads()` should reproduce the original object. Across all FSSs, 37 tests referenced this requirement, of which 24 (65%) correctly verified it. In contrast, 103 tests referenced it in all available LPSs, but only 39 (38%) were truly aligned with the specified behavior.

This inflation indicates that, as new edge case tests are generated during refinement, they are often assigned to this broadly defined requirement because no more specific one applies. Thus, ENC-CORE-1 seems to have become a "catch-all" category for unlinked tests, reinforcing the interpretation that LIFT struggles to maintain semantic precision when expanding beyond the documented requirement space.

```python
@pytest.mark.reporting(
    developer="automatic",
    functional_specification="ENC-CORE-1",
    test_description="dumps returns str and round-trips via loads"
)
@pytest.mark.category("unit")
def test_dumps_returns_str_and_roundtrips():
    obj = {"a": [1, 2, 3], "b": {"c": "text"}}
    s = json.dumps(obj)
    assert isinstance(s, str), f"dumps should return str, got {type(s)!r}"
    out = json.loads(s)
    assert out == obj, f"loads(dumps(obj)) must equal original object; got
     ↪ {out!r}"
```

Listing 5: Example for a test covering requirement ENC-CODE-1

Similar to Listing 3, Listing 5 highlights that a short and precise unit test can cover the ENC-CORE-1 acceptance criterion sufficiently. In practice, most test suites include one or more tests just like this. Yet, Listing 5 also represents a misaligned test. The requirement ENC-CORE-1 states that a JSON string shall be generated. Its acceptance specifies the `str` type check as well as the comparison between the original Python object and the object retrieved after the round-trip. The test implements both parts of the acceptance criteria, but overlooks a check of the generated string itself. The mismatched requirement acceptance can be identified as the origin of this behavior, but human reasoning would expect that the generated string would also be checked for the correct content. The

current state of the unit test does verify the round-trip behavior; the PUT's output could have been in any different format and the underlying defect would not be revealed by this test - highlighting a flaw in the assertions quality that is representative for the generated tests throughout all test suites.

From a qualitative standpoint, the generated tests are generally short, descriptive, and syntactically well-formed. Function and variable names are clear and the `pytest` markers needed for proper requirements linking and other functionality like conditional skipping of tests are consistently applied in accordance with framework conventions. However, as already discussed in Subsection 6.2.3, the overall file organization deteriorates over iterations. New tests are frequently written into newly created files rather than integrated into existing clusters, preventing meaningful grouping or modular organization. As a result, despite the test's individual readability, the global structure of the generated test suites is fragmented and difficult to maintain.

6.2.7 Integration and System Test Behavior

Across the 24 conducted trials, only one FSS and three LPSs contained a total of three and 13 tests respectively that were labeled by the TSG as system tests. All of these tests targeted the `cli` requirements `CLI-1` ("Pretty-print tool"), `CLI-2` ("CLI error reporting"), and `CLI-3` ("CLI default arguments")[14]. This observation reinforces the assumption that the internal labeling of test types is inconsistent, as all test suites included comparable tests addressing the same `cli`-related behavior. Since the coverage results confirm that these parts of the PUT were executed in every suite, the absence of system test labels cannot be attributed to missing functionality but rather to inconsistent classification within the generation process.

Notably, the tests implementing this behavior were frequently multi-labeled as both integration and system tests, which is reasonable given that command-line invocations naturally span multiple components and represent end-to-end execution paths. In test suites without any system tests, the only integration tests were again those covering the CLI functionality, further indicating that the TSG associates higher-level testing predominantly with the visible entry points of the application. Beyond these, no coherent clusters of integration tests were identified. Potential integration targets, such as verifying the interaction between the high-level `dump/s()` and `load/s()` functions and their underlying `JSONEncoder/JSONDecoder` components, remained untested, likely because such relationships were not explicitly described in the given requirements and thus never inferred by the model. Additionally, the test types to implement were only mentioned in the `pytest` configuration available to the LLM agents. No clear definition of the test types and instruction on how and when to use them was provided to the agents in their system prompts.

From these observations, two key insights emerge. Firstly, the requirements specification appears underspecified regarding integration and system behavior, providing no structured pathway for constructing higher-level tests. Secondly, the LLMs exhibit difficulties in consistently distinguishing between test levels, as seen in the recurring mislabeling of the CLI-related tests. Together, these findings suggest that while the generated test suites are capable of exercising cross-module behavior, the concept of hierarchical testing is not yet robustly internalized by the model. Higher-level interactions are only explored when they coincide with clearly exposed functionality, such as the Command-Line Interface, leaving deeper integration scenarios largely uncovered.

[14] Find the complete requirements in Appendix C.

***RQ-2.4*: How well do LLM-generated test suites fulfill traditional and possible LLM-adapted quantitative and qualitative metrics?**

The evaluation of LIFT shows that LLM-generated test suites can largely satisfy traditional quantitative metrics of test quality, while qualitative aspects remain only partially fulfilled. In terms of correctness, LIFT achieves a 100% success rate for all generated suites once environment dependencies are resolved. This represents a substantial improvement over prior approaches such as ChatTester (41% passing tests) [38] and TestART (78.6% passing tests) [15], demonstrating that full syntactic and semantic validity can be maintained across iterative refinements.
Regarding structural sufficiency, the generated suites steadily expand and reach line coverage levels comparable to, or slightly exceeding, the human-written baseline, though branch coverage remains lower. This indicates that LLMs can achieve broad statement level adequacy but still struggle to explore complex conditional logic. The mutation scores ($\approx$ 60% for LPS) further confirm that, despite notable improvement from earlier iterations ($\approx$ 44% for FSS), fault-detection capacity seems to lag behind the original suite (73%), consistent with previous findings that coverage and effectiveness correlate only weakly [7]. Thus, LIFT achieves quantitative soundness but still exhibits a semantic gap in behavioral depth.
From a qualitative perspective, the generated tests are syntactically clear and employ meaningful naming and consistent marker usage, but the overall suite organization deteriorates across iterations. Unclear test type labeling, fragmented file structures, and an increasing share of hallucinated requirement identifiers ($\approx$ 24% in final suites) reduce maintainability and traceability. Nonetheless, individual test readability remains high, aligning with ChatTester's observation that passing LLM-generated tests can resemble human-written ones [38].

Overall, LIFT fulfills the quantitative criteria of correctness, coverage, and test suite completeness to a high degree, surpassing previous LLM-based methods. However, performance in qualitative dimensions like maintainability, semantic precision, and requirement grounding remains limited. These results suggest that while LLM-generated suites can reach structural adequacy comparable to traditional methods, they still require improved internal consistency and contextual reasoning to achieve full equivalence to manually engineered test suites.
From the presented experiments, only a general indication of current LLM capabilities can be inferred. It is plausible that larger models or more precisely defined requirements could further enhance test suite effectiveness. Conversely, the selected PUT, `simplejson`, implements widely known and well-documented JSON functionality that frequently appears in the training corpora of modern LLMs. This likely contributes to the comparatively strong results observed. Consequently, the reported performance could also be interpreted as an upper bound and worse outcomes may be expected for less common or domain-specific software systems.

***RQ-2.5*: To what extent are LLM-generated test suites able to detect faults beyond simply achieving high coverage?**

The results show that high structural coverage of LLM-generated test suites does not necessarily translate into equally strong fault-detection capability. While LIFT achieves average line coverage levels comparable to the human-written reference suite, its mutation scores remain notably lower ($\approx$ 60% vs. 73%). This discrepancy indicates that the generated tests exercise large portions of the code but often fail to assert semantically relevant behavior. Similar findings have been confirmed in mutation-driven studies such as MuTAP [10], which observed only a weak correlation between coverage and test effectiveness reported by Cai and Lyu [7].
In LIFT, the refinement process improves mutation performance over iterations (from 44% for the FSSs to 60% for the LPSs), demonstrating that iterative feedback helps sharpen test assertions and enhance behavioral depth. Nevertheless, the remaining gap to `simplejson`'s original suite highlights that the generated assertions are still shallow, capturing expected syntax or output structure rather than program semantics.
These results suggest that LLM-generated suites are capable of achieving structural completeness but only partially capture the behavioral diversity required for strong fault revelation. High coverage, therefore, reflects statement execution broadness rather than robust behavioral validation. LIFT attempts to integrate semantic feedback through its TSE, which highlights requirement gaps and advises on meaningful extensions. Further improvements in the depth and specificity appear to be needed to bridge the remaining gap and enable the detection of subtle or logic-level errors beyond what coverage metrics alone can reveal.

_RQ-2_: Are LLMs able to generate "good" test suites?

Good test suites are characterized as not only being executable but also exhibiting strong quantitative and qualitative characteristics of effectiveness. Following the criteria established in Subsection 2.3 and Subsection 6.1, the evaluation of the proposed LIFT framework reveals that LLMs are indeed capable of producing syntactically correct, consistently executable, and quantitatively strong test suites, yet still fall short in qualitative depth and semantic precision.
Quantitatively, the results show that all generated test suites achieve full syntactic correctness, thereby clearly surpassing prior approaches such as *ChatTester* [38] (41% pass rate) and *TestART* [15] (78.6% pass rate). Line coverage reaches or slightly exceeds the level of the human reference suite, while branch coverage remains noticeably lower. Mutation testing further indicates a steady improvement of the generated suites over the course of the iterations. The mean mutation score (*MS*) increases from roughly 44% for the First Sufficient Test Suites (FSSs) to 60% for the Last Passing Test Suites (LPSs), whereas the manually created baseline attains 73%. This demonstrates that the iterative process successfully refines the behavioral adequacy of the suites, even though the absolute fault-detection capability still lags behind that of human engineers.
From a qualitative perspective, the generated tests are generally well structured and readable, featuring clear naming, coherent formatting, and logically organized assertions. However, qualitative inspection also reveals a certain degree of fragmentation, mislabeling of the test type, and inconsistency across iterations. While individual tests appear plausible and purposeful, their aggregation into a cohesive and maintainable suite remains challenging. In particular, the final iterations exhibit a non-negligible proportion of hallucinated requirement identifiers ($\approx 24\%$) and occasional redundancy among test cases. Consequently, the suites show a high degree of syntactic maturity but limited semantic grounding, as many tests focus on executing code paths rather than verifying meaningful behavioral properties.
Taken together, these findings suggest that LLMs can already generate test suites that are quantitatively *good* in the traditional sense of test quality. Their ability to reason about the underlying program logic and to design assertion-rich, semantically meaningful tests, however, remains limited - as already indicated by Fan et al. [12]. The gap between structural and behavioral adequacy thereby delineates the current frontier of LLM-based testing. Future advancements in model reasoning, prompt design, and feedback integration mechanisms may help bridge this gap and enable LLMs to reach parity with human-crafted test suites in both quantitative and qualitative terms.

In summary, LLMs are demonstrably capable of producing *good* test suites when assessed by structural metrics and syntactic soundness. Yet, they remain qualitatively imperfect - lacking the semantic depth, consistency, and maintainability that define a truly human-engineered suite. Nonetheless, the observed trajectory of improvement highlights their growing potential and underlines the promise of iterative, feedback-driven approaches such as LIFT to further enhance the quality and completeness of automatically generated test suites, already surpassing traditional automated approaches.

6.3 Threats to validity

The validity of the obtained results depends on the soundness of the experimental design and the generalizability of its observations. Following established guidelines in Software Engineering research [27, 34], potential threats to validity are discussed along two primary dimensions: internal validity, addressing influences that may distort the causal interpretation of the results, and external validity, concerning the extent to which the findings can be generalized beyond the studied context. The following subsections outline the main limitations identified for this work and describe their expected impact on the interpretation of the results.

6.3.1 Internal Validity

The internal validity of this study may be affected by several design and methodological constraints that can influence the interpretation of its results. One primary threat arises from the exclusion of broken trials from the final analysis. While this decision ensures that only complete data are evaluated, it potentially biases results toward stability, as failing trials that collapsed prematurely are omitted entirely. Including these trials will result in reduced success rates and increased variance across trials, yet would more accurately reflect the underlying volatility of the system. This selective exclusion thus represents a trade-off between data integrity and representativeness.

A second source of threat stems from the imposed limit of 25 iterations per trial. Although the results indicate continuous progress throughout this range, it remains uncertain whether all trials would eventually converge toward a sufficient test suite (FSS) if allowed to continue. This limitation constrains conclusions about long-term convergence and might mask late-stage stabilization patterns that would emerge under extended optimization runs.

Furthermore, the sufficiency criterion defined in the Test Suite Evaluator prompt lacks formal specification. The concept of a "sufficient" test suite is inherently qualitative and relies on the model's internal judgment rather than a consistent metric. This imprecision contributes to unstable or inconsistent stopping conditions and contradicts the proclaimed overall reliability of LLM-as-a-Judge evaluation methods [16, 42], yet is in line with the reported instability of single answer grading [42, p. 4]. Consequently, the determination of completion may vary arbitrarily across trials, affecting comparability and internal consistency.

The apparent abstraction level of the behavioral requirements represents a further threat to internal validity, since many requirements are expressed at a high level of granularity. The TSE may identify untested behavioral gaps that are not explicitly stated in any single requirement. This seems to result in requirement hallucinations, where the model introduces tests that can not be linked to any high-level requirement correctly. According to Pressman, such specification gaps can induce untraceable error sources in validation activities, as they blur the boundary between intended and emergent functionality [27]. Similarly, Sommerville highlights that inadequate requirement precision directly undermines the verifiability of system behavior [34].

Additionally, the unclear definition of the test type required to test a requirement, combined with the absence of explicit specification of test types in the system prompts of the TSG and TSE, appears to lead to mislabeling of test levels. Therefore, the real composition of the test suites remains hidden.

Finally, the selection bias introduced by aggregating only successful iterations (i.e., excluding error-/failure-producing test suite iterations) poses another threat. While this approach prevents erratic data points from skewing aggregate measures, it may also underrepresent phases of instability or exploration that are intrinsic to iterative optimization. This effect is mitigated by focusing on the last produced test suite state (LPS) of each trial, which reflects a relevant output of the LIFT process. Nonetheless, the cumulative effect of these threats should be considered when interpreting overall trends and causal attributions within the evaluation.

6.3.2 External Validity

The external validity of the study is limited by several factors that constrain the generalizability of its findings beyond the experimental setting. First and foremost, all experiments were conducted on a single Program under Test (`simplejson`). While this library provides a well-defined and self-contained domain for observing test suite evolution, the results may not directly transfer to projects of a different nature, size, or complexity. As Pressman emphasizes, software processes and quality outcomes are strongly context-dependent, varying significantly across application domains and development environments [27]. Future studies should therefore replicate the LIFT process across multiple PUTs to validate the robustness of the observed patterns.

A related limitation arises from the structural simplicity of the chosen PUT. The `simplejson` library primarily exposes functional units with limited inter-module interaction, which restricts the generation of higher-level test cases, such as integration or system tests. More complex systems would naturally involve richer dependencies, shared state, and multi-component behavior elements that could better reveal the boundaries of LLM-based testing capabilities. Extending the evaluation to projects with broader architectural diversity would thus enhance the generality of the findings.

Another constraint concerns the technological stack. All experiments were executed within a Python environment, which differs substantially from statically typed or compiled ecosystems such as Java or C#. These languages employ distinct build, dependency, and testing infrastructures that may affect both toolchain behavior and LLM comprehension of code semantics. The cross-language transferability of the approach, therefore, remains uncertain. To address this, future replications should examine compiled languages and heterogeneous environments, thereby testing whether LIFT's results hold across different Software Engineering ecosystems.

Finally, scalability remains a central external limitation. The token limits inherent to current LLM architectures impose an upper bound on the size of analysable PUTs, constraining the evaluation to relatively small or moderately complex projects. Consequently, the ability of LIFT to operate on large-scale repositories, industrial codebases, or multi-repository systems remains unverified. Future research should therefore investigate larger PUTs and leverage different models and model families with extended context windows or specialized retrieval mechanisms to assess whether the observed behavior persists at scale.

Overall, while the presented results provide a strong proof of concept, their external validity is inherently limited to small, Python-based projects. Broader replication across systems, languages, and model architectures is required to substantiate general claims about the generalizability of LIFT's performance and behavior.

7 Future Work

The LIFT approach presented in Section 4 and the achieved results in Subsection 6.2 demonstrate the feasibility of iterative test suite generation and highlight several avenues for further exploration. Future work should aim to enhance both the robustness and generalizability of the approach while addressing the identified threats to internal and external validity.

Firstly, the external validity can be strengthened by evaluating the framework on a broader set of Programs under Test. While the current study focused on a single library (`simplejson`), replication across projects of varying domains, sizes, and languages will be crucial to establish generalizability. Extending LIFT to other ecosystems, such as Java or C#, would also test its adaptability to compiled environments, thereby directly addressing the limitations imposed by the current Python-only setup.

Second, future work should investigate scalability with respect to both project complexity and model size. Current experiments were constrained by model context length and token limits, which limited the size of analysable repositories and the amount of conversational history available to the agents. Employing models with extended context windows or retrieval-augmented architectures could enable processing of larger codebases and preserve iterative learning across generations. Providing the LLM with structured access to prior iterations could allow for cumulative reasoning and long-term refinement capabilities that mirror gradient accumulation in numerical optimization.

Third, internal validity may be improved through a refined experimental design. The current implementation employs a unified generator system prompt agent with instruction-based differentiation between initial and refinement phases. Separating these roles into distinct agents with specialized prompts and system instructions could stabilize iterative behavior. Additionally, the introduction of more sophisticated, task-specific tool interfaces for code navigation, test execution, and feedback interpretation will likely help the LLM to reason more effectively about code changes and test outcomes.

Furthermore, prompt engineering remains a key lever for performance improvement. This could include better ways to handle misalignment in requirements linking and under complex assertion quality. Adding instructions to specifically check for these occurrences and introducing the option to not link tests to any requirement if checked behavior is emergent and unspecified could improve traceability.

Lastly, the experimental setup can be extended with more complex conversation management strategies. Instead of providing the complete message history within the agent, future implementations may selectively include only the latest file state or a summary of relevant changes. This would mitigate context overload while preserving continuity of reasoning. Conversely, pre-loading key files at the beginning of a conversation could allow the agent to avoid repeated re-establishment of its understanding of the environment and the PUT. This would eliminate the *Project Discovery Phase* repeated by each agent for each iteration and could save tokens.

Overall, these directions not only aim to improve the technical capabilities of LIFT but also to mitigate several of the threats identified in Subsection 6.3. Broader replication, refined iteration management, and enhanced context handling will contribute to greater reliability, scalability, and generalizability.

8 Conclusion

The increasing integration of Large Language Models (LLMs) into Software Engineering tasks has opened new possibilities for automation in quality assurance, particularly within Software Testing. Recent research demonstrates that LLMs are not only capable of generating syntactically correct code but can also create meaningful test cases for a variety of programming tasks [15, 38]. Building on these advances, this thesis explores the capability of LLMs to autonomously construct and refine complete test suites for real-world software projects.

The thesis positions itself within the ongoing evolution of LLM-driven testing approaches, highlighting key developments such as *TextGrad* (Subsection 3.3) and *LLM-as-a-Judge* (Subsection 3.4). *TextGrad* introduces the concept of textual gradients, framing LLM feedback as a differentiable optimization process, while *LLM-as-a-Judge* establishes the models' ability to evaluate and critique their own or other models' outputs. Both concepts demonstrate an emerging form of iterative self-improvement that extends beyond static code generation.

Building on these foundations, this work introduces LLM-based Iterative Feedback-driven Test suite generation (LIFT) (Section 4), a novel framework that applies the principles of textual differentiation and self-evaluation to the automated generation of whole test suites. The aim is to examine whether LLMs can progress from one-shot test generation toward an iterative process capable of producing complete, executable, and improving test suites without human intervention. Through this, the thesis contributes to a deeper understanding of the current capabilities and limitations of LLMs within the domain of Software Testing, and provides empirical insights into their potential as autonomous testing agents.

Main Findings The findings of this thesis in addition to prior research [10, 15] confirm that LLMs are capable of autonomously generating complete and executable test suites, positively answering RQ-1 - *"Are LLMs able to generate test suites at all?"*. **LIFT successfully produced syntactically valid and runnable test suites** that interacted meaningfully with the underlying Program under Test. This supports earlier results reported by Yuan et al. in [38], Chen et al. in [9], and Gu et al. in [15], who demonstrated the ability of LLMs to produce executable unit tests for diverse projects and languages. Similarly, approaches such as *MuTAP* [10] have shown that the combination of generation and automatic repair can further enhance the reliability of LLM-produced tests. The consistent generation of functional test code observed in this work confirms that LLMs have reached a level of maturity that allows them to produce not only individual tests, but entire runnable suites.

The answer to RQ-2 - *"Are LLMs able to generate "good" test suites?"* is more nuanced. Through iterative feedback, LIFT demonstrated a steady improvement in test suite size and measurable quality metrics. On average, both line and branch coverage increased throughout the observed iterations, reaching an **average line coverage** of 75.60% for First Sufficient Test Suites (FSSs) and **83.84% for Last Passing Test Suites (LPSs)** as well as **branch coverages** of 68.75% and **75.91%** respectively. Mutation testing revealed improvements on par with other approaches, yet the LIFT-generated suites only achieve **lower mutation scores** compared to their human-written counterpart.

Nevertheless, the results also reveal limitations in qualitative refinement. While the framework consistently extends existing suites with new tests, it shows a **tendency to prioritise quantity over diversity** and to replicate similar behavioral patterns. In some cases, hallucinated inputs or

redundant assertions suggest that the refinement process lacks a genuine understanding of semantic adequacy.

Despite these limitations, the observed progress across iterations demonstrates that the textual feedback mechanism enables meaningful incremental improvement without manual guidance. This positions LIFT as the first framework to apply the concept of textual differentiation to the domain of Software Testing. By combining self-evaluative feedback with automated test execution and mutation-based validation, LIFT establishes an empirical foundation for LLM-driven test-suite evolution. It therefore contributes two major advances:

1. the design of an iterative, feedback-driven test generation framework based on textual gradients and
2. the demonstration that LLMs can autonomously evolve complete and executable test suites.

Limitations and Future Work The results presented in this thesis are subject to **several limitations that constrain their generalisability**. The most prominent threat to internal validity arises from the exclusion of broken or incomplete trials in the final analysis, which may lead to a slight overestimation of average performance. In addition, the inherent stochasticity of LLM behavior can affect reproducibility, as small prompt or sampling variations can yield divergent outcomes. The external validity of the results is limited by the choice of a single PUT and a Python-only environment. While the selected project provides a representative yet manageable code base, it does not reflect the full diversity of software systems, languages or testing frameworks found in practice. Furthermore, the token limitations of the employed models constrained both the project size and the depth of iterative context that could be preserved, restricting the scalability of the approach.

Future work should address these limitations by extending the experimental scope to include a wider range of projects, programming languages, and testing paradigms. Scaling to larger PUTs and more capable LLMs with extended context windows could allow for richer interaction histories and deeper iterative refinement. Another promising direction is the definition of specialised tool functions within LIFT and the sharpening of agent roles through prompt engineering. Integrating persistent conversational memory could further improve coherence between iterations and reduce redundant or hallucinated test cases.

To the author's knowledge, this is the first work not only to attempt to generate full test suites but also analyse their quality holistically. The evaluation points towards a future direction of research. As LLMs become more capable in fulfilling numerical metrics like coverage or mutation scores, the next important challenge is to ensure that the created tests are meaningful and not superficial. Future research into an automated analysis of this quality dimension seems to be a cornerstone of upcoming advances in the field of LLM-based test generation.

In summary, this thesis demonstrates that LLMs can already act as autonomous agents capable of producing and iteratively refining executable test suites. While the current state of technology still imposes practical limitations, the achieved results indicate a **trajectory towards increasingly independent, self-improving testing systems**. The proposed LIFT framework represents an initial but significant step in this direction, bridging the gap between generative code synthesis and systematic software validation. As future models grow in reasoning ability and context capacity, their integration into established Software Engineering processes will further blur the boundaries between human-guided and machine-driven testing, moving the field closer to genuinely autonomous software quality assurance.

References

[1] M. Alenezi and M. Akour. "AI-Driven Innovations in Software Engineering: A Review of Current Practices and Future Directions". In: *Applied Sciences* 15.3 (Jan. 2025), p. 1344. ISSN: 2076-3417. DOI: 10.3390/app15031344. URL: https://www.mdpi.com/2076-3417/15/3/1344 (visited on 09/17/2025).

[2] K. Beck et al. *Manifesto for Agile Software Development*. Manifesto for Agile Software Development. 2001. URL: https://agilemanifesto.org/ (visited on 09/20/2025).

[3] L. Belzner, T. Gabor, and M. Wirsing. "Large Language Model Assisted Software Engineering: Prospects, Challenges, and a Case Study". In: *Bridging the Gap Between AI and Reality*. Ed. by B. Steffen. Vol. 14380. Cham: Springer Nature Switzerland, 2024, pp. 355–374. DOI: 10.1007/978-3-031-46002-9_23. URL: https://link.springer.com/10.1007/978-3-031-46002-9_23 (visited on 10/21/2025).

[4] B. W. Boehm. "A Spiral Model of Software Development and Enhancement". In: *Computer* 21.5 (May 1988), pp. 61–72. ISSN: 0018-9162. DOI: 10.1109/2.59. URL: http://ieeexplore.ieee.org/document/59/ (visited on 09/20/2025).

[5] P. Bourque and R. E. Fairley, eds. *SWEBOK: Guide to the Software Engineering Body of Knowledge*. Version 3.0. Los Alamitos, CA: IEEE Computer Society, 2014. ISBN: 978-0-7695-5166-1.

[6] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC Editor, Dec. 2017. DOI: 10.17487/RFC8259. URL: https://www.rfc-editor.org/info/rfc8259.

[7] X. Cai and M. R. Lyu. "The Effect of Code Coverage on Fault Detection under Different Testing Profiles". In: *ACM SIGSOFT Software Engineering Notes* 30.4 (July 2005), pp. 1–7. ISSN: 0163-5948. DOI: 10.1145/1082983.1083288. URL: https://dl.acm.org/doi/10.1145/1082983.1083288 (visited on 10/30/2025).

[8] T. T. Chekam et al. "An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). May 2017, pp. 597–608. DOI: 10.1109/ICSE.2017.61. URL: https://ieeexplore.ieee.org/document/7985697 (visited on 09/10/2025).

[9] Y. Chen et al. *ChatUniTest: A Framework for LLM-Based Test Generation*. May 7, 2024. DOI: 10.48550/arXiv.2305.04764. arXiv: 2305.04764 [cs]. URL: http://arxiv.org/abs/2305.04764 (visited on 10/06/2025). Pre-published.

[10] A. M. Dakhel et al. *Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing*. Aug. 31, 2023. DOI: 10.48550/arXiv.2308.16557. arXiv: 2308.16557 [cs]. URL: http://arxiv.org/abs/2308.16557 (visited on 09/10/2025). Pre-published.

[11] Divyani Shivkumar Taley. "Comprehensive Study of Software Testing Techniques and Strategies: A Review". In: *International Journal of Engineering Research and Technology (IJERT)* V9.08 (Sept. 4, 2020), IJERTV9IS080373. ISSN: 2278-0181. DOI: 10.17577/IJERTV9IS080373. URL: https://www.ijert.org/comprehensive-study-of-software-testing-techniques-and-strategies-a-review (visited on 09/17/2025).

[12] A. Fan et al. *Large Language Models for Software Engineering: Survey and Open Problems*. Nov. 11, 2023. DOI: 10.48550/arXiv.2310.03533. arXiv: 2310.03533 [cs]. URL: http://arxiv.org/abs/2310.03533 (visited on 09/10/2025). Pre-published.

[13] M. S. Farooq et al. "Behavior Driven Development: A Systematic Literature Review". In: *IEEE Access* 11 (2023), pp. 88008–88024. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3302356. URL: https://ieeexplore.ieee.org/document/10210040/ (visited on 11/08/2025).

[14] G. Fraser and A. Arcuri. "A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite". In: *ACM Transactions on Software Engineering and Methodology* 24.2 (Dec. 23, 2014), pp. 1–42. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/2685612. URL: https://dl.acm.org/doi/10.1145/2685612 (visited on 11/02/2025).

[15] S. Gu et al. *TestART: Improving LLM-based Unit Testing via Co-evolution of Automated Generation and Repair Iteration*. Mar. 31, 2025. DOI: 10.48550/arXiv.2408.03095. arXiv: 2408.03095 [cs]. URL: http://arxiv.org/abs/2408.03095 (visited on 09/24/2025). Pre-published.

[16] J. He et al. *LLM-as-a-Judge for Software Engineering: Literature Review, Vision, and the Road Ahead*. Oct. 28, 2025. DOI: 10.48550/arXiv.2510.24367. arXiv: 2510.24367 [cs]. URL: http://arxiv.org/abs/2510.24367 (visited on 11/02/2025). Pre-published.

[17] J. Hoffmann et al. *Training Compute-Optimal Large Language Models*. Mar. 29, 2022. DOI: 10.48550/arXiv.2203.15556. arXiv: 2203.15556 [cs]. URL: http://arxiv.org/abs/2203.15556 (visited on 11/08/2025). Pre-published.

[18] X. Hou et al. *Large Language Models for Software Engineering: A Systematic Literature Review*. Apr. 10, 2024. DOI: 10.48550/arXiv.2308.10620. arXiv: 2308.10620 [cs]. URL: http://arxiv.org/abs/2308.10620 (visited on 10/31/2025). Pre-published.

[19] J. Humble and D. G. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 7th printing 2013. A Martin Fowler Signature Book. Upper Saddle River, NJ: Addison-Wesley, 2011. 463 pp. ISBN: 978-0-321-60191-9.

[20] B. Ippolito. *Simplejson/Simplejson*. Aug. 17, 2025. URL: https://github.com/simplejson/simplejson (visited on 08/22/2025).

[21] R. Just and D. Jalali. *Rjust/Defects4j*. Oct. 17, 2025. URL: https://github.com/rjust/defects4j (visited on 10/21/2025).

[22] A. Madaan et al. *Self-Refine: Iterative Refinement with Self-Feedback*. May 25, 2023. DOI: 10.48550/arXiv.2303.17651. arXiv: 2303.17651 [cs]. URL: http://arxiv.org/abs/2303.17651 (visited on 09/10/2025). Pre-published.

[23] A. S. Namin and J. H. Andrews. "The Influence of Size and Coverage on Test Suite Effectiveness". In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ISSTA '09: International Symposium on Software Testing and Analysis. Chicago IL USA: ACM, July 19, 2009, pp. 57–68. ISBN: 978-1-60558-338-9. DOI: 10.1145/1572272.1572280. URL: https://dl.acm.org/doi/10.1145/1572272.1572280 (visited on 10/05/2025).

[24] NASA. *NASA Systems Engineering Handbook*. NASA, 2018. URL: https://www.nasa.gov/wp-content/uploads/2018/09/nasa_systems_engineering_handbook_0.pdf.

[25] S. Nugroho, S. H. Waluyo, and L. Hakim. "Comparative Analysis of Software Development Methods between Parallel, V-Shaped and Iterative". In: *International Journal of Computer Applications* 169.11 (July 17, 2017), pp. 7–11. ISSN: 09758887. DOI: 10.5120/ijca2017914605. arXiv: 1710.07014 [cs]. URL: http://arxiv.org/abs/1710.07014 (visited on 09/18/2025).

[26] D. Paul et al. *REFINER: Reasoning Feedback on Intermediate Representations*. Feb. 4, 2024. DOI: 10.48550/arXiv.2304.01904. arXiv: 2304.01904 [cs]. URL: http://arxiv.org/abs/2304.01904 (visited on 09/10/2025). Pre-published.

[27] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. 7th ed. Dubuque, IA: McGraw-Hill, 2010. 895 pp. ISBN: 978-0-07-337597-7.

[28] W. W. Royce. "Managing the Development of Large Software Systems (1970)". In: *Ideas That Created the Future*. Ed. by H. R. Lewis. The MIT Press, Feb. 2, 2021, pp. 321–332. ISBN: 978-0-262-36317-4. DOI: 10.7551/mitpress/12274.003.0035. URL: https://direct.mit.edu/books/book/5003/chapter/2657056/Managing-the-Development-of-Large-Software-Systems (visited on 09/18/2025).

[29] R. Santos et al. *Are We Testing or Being Tested? Exploring the Practical Applications of Large Language Models in Software Testing*. Dec. 8, 2023. DOI: 10.48550/arXiv.2312.04860. arXiv: 2312.04860 [cs]. URL: http://arxiv.org/abs/2312.04860 (visited on 09/10/2025). Pre-published.

[30] M. Schäfer et al. *An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation*. Dec. 11, 2023. DOI: 10.48550/arXiv.2302.06527. arXiv: 2302.06527 [cs]. URL: http://arxiv.org/abs/2302.06527 (visited on 09/10/2025). Pre-published.

[31] J. Sevilla. *Training Compute of Frontier AI Models Grows by 4-5x per Year*. Epoch AI. May 28, 2024. URL: https://epoch.ai/blog/training-compute-of-frontier-ai-models-grows-by-4-5x-per-year (visited on 11/02/2025).

[32] J. Shin et al. *Domain Adaptation for Code Model-based Unit Test Case Generation*. July 30, 2024. DOI: 10.48550/arXiv.2308.08033. arXiv: 2308.08033 [cs]. URL: http://arxiv.org/abs/2308.08033 (visited on 10/31/2025). Pre-published.

[33] M. L. Siddiq et al. "Using Large Language Models to Generate JUnit Tests: An Empirical Study". In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. June 18, 2024, pp. 313–322. DOI: 10.1145/3661167.3661216. arXiv: 2305.00418 [cs]. URL: http://arxiv.org/abs/2305.00418 (visited on 10/31/2025).

[34] I. Sommerville. *Software Engineering*. Tenth edition. Always Learning. Boston Columbus Indianapolis New York San Francisco Hoboken Amsterdam Cape Town Dubai London: Pearson, 2016. 1 p.

[35] H. K. V. Tran et al. “Quality Attributes of Test Cases and Test Suites – Importance & Challenges from Practitioners’ Perspectives”. In: *Software Quality Journal* 33.1 (Mar. 2025), p. 9. ISSN: 0963-9314, 1573-1367. DOI: 10.1007/s11219-024-09698-w. arXiv: 2507.06343 [cs]. URL: http://arxiv.org/abs/2507.06343 (visited on 10/05/2025).

[36] J. Wang et al. *Software Testing with Large Language Models: Survey, Landscape, and Vision*. Mar. 4, 2024. DOI: 10.48550/arXiv.2307.07221. arXiv: 2307.07221 [cs]. URL: http://arxiv.org/abs/2307.07221 (visited on 09/10/2025). Pre-published.

[37] Z. Wang et al. *HITS: High-coverage LLM-based Unit Test Generation via Method Slicing*. Aug. 21, 2024. DOI: 10.48550/arXiv.2408.11324. arXiv: 2408.11324 [cs]. URL: http://arxiv.org/abs/2408.11324 (visited on 10/21/2025). Pre-published.

[38] Z. Yuan et al. *No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation*. May 19, 2024. DOI: 10.48550/arXiv.2305.04207. arXiv: 2305.04207 [cs]. URL: http://arxiv.org/abs/2305.04207 (visited on 09/10/2025). Pre-published.

[39] M. Yuksekgonul et al. *TextGrad: Automatic "Differentiation" via Text*. June 11, 2024. DOI: 10.48550/arXiv.2406.07496. arXiv: 2406.07496 [cs]. URL: http://arxiv.org/abs/2406.07496 (visited on 09/10/2025). Pre-published.

[40] P. Zhang et al. *Test Suite Effectiveness Metric Evaluation: What Do We Know and What Should We Do?* Apr. 19, 2022. DOI: 10.48550/arXiv.2204.09165. arXiv: 2204.09165 [cs]. URL: http://arxiv.org/abs/2204.09165 (visited on 10/05/2025). Pre-published.

[41] Y. Zhang and A. Mesbah. “Assertions Are Strongly Correlated with Test Suite Effectiveness”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE’15: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. Bergamo Italy: ACM, Aug. 30, 2015, pp. 214–224. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786858. URL: https://dl.acm.org/doi/10.1145/2786805.2786858 (visited on 10/05/2025).

[42] L. Zheng et al. *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena*. Dec. 24, 2023. DOI: 10.48550/arXiv.2306.05685. arXiv: 2306.05685 [cs]. URL: http://arxiv.org/abs/2306.05685 (visited on 10/06/2025). Pre-published.

A LIFT System Prompts

A.1 Test Suite Generator System Prompt

```
# Your Role
You are a Python Test Suite Generation Expert with 15 years of experience specializing in
↪ creating comprehensive pytest test suites for Python projects. You create and refine test
↪ suites for a given Python project based on behavioral requirements against this project
↪ and the feedback of experts evaluating the behavior of the test suite. If the feedback and
↪ requirements contradict each other, follow the given feedback and neglect the requirements.

# Current Environment
   - OS: Win10 x64
   - Python: 3.12.10
   - pytest: 8.3.5

# Your Workflow
1. **Project Discovery Phase**
    - Understand the current folder structure
    - Identify all Python source files in the project
    - Locate existing test files (if any) in `tests/` directory

2. **Comprehensive Understanding Phase**
   a) **Source Code Analysis**:
    - Read all Python source files to understand functionality
    - Understand data flow and dependencies between modules

   b) **Test Environment Configuration**:
    - Read `pytest_html_report.yml` to understand test configuration
    - Note any special fixtures, plugins, or settings
    - Identify test discovery patterns and conventions
    - Remember the test categories

3. **Requirements Gathering Phase**
    - Read `program-requirements.yml` to understand the program requirements
    - Understand the requirements for the program behavior
    - Identify, how to test each requirement (test type, acceptance requirement, etc.)
    - Link the `functional_specs` of the `pytest_html_report.yml` to the requirements in the
    ↪  `program-requirements.yml`

4. **Feedback Analysis Phase** (Conditional - for refinement iterations)
   a) **Observations (xml-reports in `/reports`)**:
    - Identify syntax errors, import errors or fixture problems
    - Understand assertion failures and their causes
    - Note any environment-specific issues

   b) **Failure Analysis (`reports/fixes.md`)**:
    - Identify the error locations needing fixes
    - Understand the root cause and the proposed changes

   c) **External Feedback (`reports/evaluation.md`)**:
    - Read the evaluation report thoroughly
    - Understand specific improvements requested
```

- Collect needed changes for better coverage

5. **Test Implementation Phase**
   a) **Fix Critical Issues First** (if any):
   - Correct all failing tests based on generated reports
   - Fix syntax/import errors preventing test execution
   - Ensure all tests can run without errors or failures
   - NEVER remove or skip existing tests if they have errors or failures (you MUST NOT AVOID them)

   b) **Cover Program Behavior**:
   - Implement tests for each behavioral requirement
   - Each requirement needs to be covered by one or multiple tests
   - Each test can cover one or multiple requirements
   - Add positive path tests to confirm expected functionality
   - Add boundary value and edge case tests derived from requirements
   - Add negative/error case tests using `pytest.raises` for defined error conditions
   - Use `pytest.mark.parametrize` for input variations

   c) **Create/Enhance Test Coverage**:
   - Add edge case and optional parameter tests
   - Test default values and parameter toggles
   - Add error condition tests that are realistically reachable
   - Guard platform-specific code with `pytest.mark.skipif` when not applicable on Win10

   d) **Test Organization**:
   - Structure tests logically (ONE test file per source file)
   - Place shared fixtures in `tests/conftest.py`
   - Keep fixtures minimal, scoped to function unless needed otherwise
   - ALL test need to follow the `TEST TEMPLATE` defined below
   - Link the covered requirements (`id` field) to the tests (`functional_specification` field)
   - Add a short description of the test behavior/goal (`test_description` field)
   - Mark each test with the applicable category (`category` field)

   e) **Quality Checks**:
   - Ensure tests are independent and can run in any order
   - Avoid mutating global/module state across tests
   - Use precise assertions with specific expected values
   - Ensure helpful error messages in assertion failures
   - Keep tests performant and avoid unnecessary delays
   - Follow existing `pytest_html_report.yml` configuration

   f) **Feedback Integration Loop** (Conditional - if `reports/evaluation.md` exist):
   - Apply all feedback items directly in tests (tests must match actual behavior)
   - Add or expand coverage as requested in evaluation
   - Re-run test suite and confirm all issues are fixed
   - Update requirement-to-test mapping comments when adding coverage

6. **Final Review**
   - Verify all feedback points have been addressed
   - Ensure no tests were accidentally deleted or broken
   - Confirm test files follow pytest conventions
   - Check that all critical paths have test coverage

```
# Expected Output
You MUST ONLY create/modify/delete files in the `tests/` folder. No other files shall be
↪  created, modified or deleted.
ONLY use the `pytest` package for your testing.

# Conversation Directives
You will not interact with the user. After the initial user request, no further input will be
↪  provided.
Don't provide any textual reasoning for your actions. Base any assumptions you make only on
↪  the input provided.
When you deem your task to be finished, you MUST send a message ONLY containing `<DONE>`.

# Critical Test Generation Principles
1. **Feedback Priority**: When `reports/evaluation.md` exists, addressing its feedback takes
↪  precedence over original requirements

2. **Failing Test Fixes**: When fixing failing tests, remember:
    - The project implementation is assumed correct
    - Tests must be adjusted to match actual project behavior
    - Don't change test logic to make it pass incorrectly
    - Base your corrections on the provided analysis

3. **Coverage Context**: Focus on meaningful coverage:
    - Don't attempt to test unreachable code
    - Skip platform-specific code that won't run on the current platform
    - Focus on testable, reachable paths

4. **Test Quality Standards**:
    - Test names should clearly indicate what is being tested
    - Use parametrize for similar tests with different inputs
    - Include both positive and negative test cases

5. **Error Handling**:
    - Test error conditions that can actually occur
    - Use pytest.raises for exception testing

# TEST TEMPLATE
Each test needs to follow this template:
```python
@pytest.mark.reporting(
    developer="automatic",
    functional_specification=<covered requirement(s) id (str or list[str])>,
    test_description=<short description of the test behavior/goal (str)>
)
@pytest.mark.category(<test category like "unit", "integration", "system" (str or list[str])>)
def <descriptive test name>(<parameters>):
    <test logic>
```
```

A.2 Test Suite Debugger System Prompt

```
# Your Role
You are a Python Test Debugger with extensive experience diagnosing Python/pytest failures.
↪ You ingest artifacts from test runs and produce, for each failure, (1) a clear error
↪ description, (2) a root-cause analysis grounded in evidence, and (3) actionable fixes —
↪ both as prose and as code patches/snippets.
You MUST assume that the program under test is correct. You will trace aerrors/failures back
↪ to wrong tests.

# Current Environment
   - OS: Win10 x64
   - Python: 3.12.10
   - pytest: 8.3.5

# Workflow
1. **Project Discovery Phase**
    - Understand the current folder structure
    - Identify all Python source files in the project
    - Locate existing test files (if any) in `tests/` directory
    - Identify all failures described in the reports (artifacts in `reports/`)

2. **Ingest & Group Failures**
    - Parse failing nodes (file::test::param), error types, messages, and tracebacks
    - Cluster related failures (same exception site/root cause) to avoid duplicate work

3. **Evidence-Driven Diagnosis**
    - For each cluster, examine stack frames, captured logs, fixtures, and the relevant test
    ↪ code
    - Cross-check implementation and configuration (plugins/marks/skipif)
    - Consider platform/reachability: distinguish untestable or platform-gated branches from
    ↪ real gaps

4. **Root-Cause Analysis**
    - Identify whether the defect resides in:^
      (a) implementation logic,
      (b) test assumptions/fixtures,
      (c) configuration/import/discovery,
      (d) platform/reachability or
      (e) environment
    - Provide minimal, reproducible reasoning (point to exact lines/conditions) with links to
    ↪ files/lines when possible
   - Assume the tested code is correct and errors are only because of incorrect ways of testing

4. **Fix Proposals (Text + Code)**
    - **Primary fix**: propose the most direct, correct change. Include a minimal patch
    ↪ (unified diff or precise snippet)
    - **Test updates**: if expectations were incorrect (e.g., spec drift), propose corrected
    ↪ assertions/fixtures/marks — never "paper over" failures (no blind skipping/xfail to
    ↪ hide defects)
    - If a path is truly untestable on this platform, justify and recommend a `skipif` guard
    ↪ instead of brittle workarounds
```

```
5. **Safety & Quality Checks**
    - Do not remove or blanket-skip failing tests
    - Prefer precise fixes that respect real behavior
    - Keep changes minimal, deterministic, and performant; maintain fixture scope hygiene and
     ↪  test independence
    - Prefer parametrization and explicit assertions in suggested test changes

6. **Output & Verdict**
    - Emit a per-error report using the template below
    - Conclude with a concise summary table of all clusters, their status, and next actions

# Expected Output
You MUST ONLY create the `reports/fixes.md` with the filled contents of the template. No other
 ↪  files shall be created, modified or deleted.
You MUST NEVER change any tests or tested code yourself!

# Conversation Directives
You will not interact with the user. After the initial user request, no further input will be
 ↪  provided.
Don't provide any textual reasoning for your actions. Base any assumptions you make only on
 ↪  the input provided.
When you deem your task to be finished, you MUST send a message ONLY containing `<DONE>`.

# Critical Principles
1. **Evidence over Intuition** - root cause must cite concrete lines/frames/artifacts
2. **Meaningful Corrections** - prefer real fixes over masking symptoms; justify any skip/xfail
3. **Contextual Reachability** - recognize platform-specific/unreachable code when deciding on
 ↪  test expectations
4. **Minimal, Safe Diffs** - smallest change that fully resolves the failure and preserves
 ↪  behavior elsewhere
5. **Traceability** - tie each fix back to the failing test(s), requirement IDs, and coverage
 ↪  implications

## Per-Error Analysis Template (use exactly this structure)
```markdown
# Error Cluster: <short name> — <exception type>
**Failing Tests:** <list of nodeids or count>
**Primary Traceback Site:** <file.py:line (function)>

**1) Error Description**
<Plain-English summary of what failed and where. Quote the key traceback line(s).>

**2) Root Cause Analysis**
- **Cause Type:** <implementation | test/fixture | config/import | platform/reachability |
 ↪  environment>
- **Evidence:** <specific lines/frames/conditions; link or quote minimal snippets>
- **Why It Fails Now:** <logic/contract mismatch, regression, wrong default, flaky timing,
 ↪  etc.>

**3) Proposed Fix (Text)**
<Explain the change and why it is correct; note side effects/risks and affected
 ↪  requirements/tests.>

**4) Proposed Fix (Code)**
```
```

```diff
<unified diff or minimal edited snippet showing the exact change(s)>
```

5) Test Impact
- **Update/Add Tests:** <what to add/change (assertions, parametrization, fixtures, marks)>
- **Coverage Note:** <does this close a meaningful gap or mark untestable/skipif>
```

---
```

A.3 Test Suite Evaluator System Prompt

```
# Your Role
You are a Test Suite Evaluation Expert with 15 years of experience specializing in assessing
↪ the quality and effectiveness of pytest test suites for Python projects. You shall
↪ identify whether the test suite is well-developed or needs enhancements. If improvements
↪ are needed, you shall provide a detailed description of missing test cases/coverage. Your
↪ most valued skill is your ability to identify whether a test suite needs reasonable
↪ refinement or is satisfactory in its current state.

# Current Environment
   - OS: Win10 x64
   - Python: 3.12.10
   - pytest: 8.3.5

# Your Workflow
1. **Project Discovery Phase**:
    - Understand the current folder structure
    - Identify all Python source files in the project
    - Locate existing test files in `tests/` directory

2. **Comprehensive Reading Phase**:
    - Read all project source files to understand the implementation
    - Read the program requirements (`program-requirements.yml`) to understand the program
    ↪ behavior
    - Read all test files to understand the test coverage
    - Read the execution and coverage reports (`reports/`) to identify possible problems and
    ↪ understand code coverage metrics

3. **Template Review**: Read the `evaluation_template.md` that you will fill with your
↪ findings.

4. **Deep Analysis Phase** (CRITICAL):
   a) **Coverage Analysis with Context**:
    - Identify uncovered lines/branches from the coverage report
    - **CRITICAL**: Evaluate whether uncovered statements are actually testable:
        * Some statements may be unreachable in the current setup (e.g. error handlers for
        ↪ conditions that can't occur)
        * Platform-specific code that won't execute on the current platform
        * Defensive programming statements that protect against impossible states
        * Debug/logging statements that may not need coverage
    - Focus on MEANINGFUL coverage gaps that represent actual missing test scenarios
    - Distinguish between "can't be covered" vs "should be covered but isn't"

   b) **Missing Test Cases Identification**:
    - Identify edge cases not tested
    - Check for boundary value testing
    - Verify error handling paths are tested (where reachable)
    - Look for integration scenarios not covered
    - Ensure that all requirements are sufficently covered

5. **Write Evaluation**: Create `reports/evaluation.md` based on the `evaluation_template.md`
↪ with:
```

```
    - Coverage gaps with context about whether they're actually testable
    - Specific, actionable recommendations for improvements
    - Clear justification for the final verdict

6. **Final Decision**: End with a message stating:
    - `<REWORK>` if the test suite has significant testable coverage gaps OR doesn't cover all
     ↪  program requirements sufficently
    - `<FINAL>` if the test suite is satisfactory (all tests pass AND coverage is adequate
     ↪  considering context)

# Expected Output
You MUST ONLY create the `reports/evaluation.md` with the filled contents of the
 ↪  `evaluation_template.md`. No other files shall be created, modified or deleted.

# Conversation Directives
You will not interact with the user. After the initial user request, no further input will be
 ↪  provided.
Don't provide any textual reasoning for your actions. Base any assumptions you make only on
 ↪  the input provided.
When you deem your task to be finished, you MUST send a message ONLY containing either
 ↪  `<REWORK>` or `<FINAL>` as described above.

# Critical Evaluation Principles
1. **Requirement Testing**: ALL requirements have to be tested sufficently.
2. **Coverage Context**: Not all uncovered code is a problem. Consider whether uncovered
 ↪  statements are actually reachable/testable in the current environment.
3. **Practical Focus**: Recommend only meaningful, achievable improvements. Don't demand 100%
 ↪  coverage if some code is genuinely untestable.
4. **Detailed Justification**: Every finding must be supported with specific examples from the
 ↪  code/reports and suggested improvements.
5. **Actionable Feedback**: Provide concrete, implementable suggestions rather than vague
 ↪  observations.
```

B LIFT Evaluation Template

```
# Test-Suite Evaluation

## 1. Coverage Snapshot

**Statement Coverage**: `<nn.%>`
**Branch Coverage**: `<nn.%>`

```text
<Explain which uncovered statements are:
- Actually testable and represent gaps
- Unreachable in the current environment (e.g., platform-specific code, error handlers for
↪  impossible conditions)
- Not requiring coverage (e.g., debug statements, defensive programming)>
```

---

2. Execution Summary

Item	Count	% of Total
Total Tests Executed	`<N>`	100 %
Passing Tests	`<N>`	`<nn.%>`
Failing Tests	`<N>`	`<nn.%>`
Skipped / Ignored	`<N>`	`<nn.%>`

Total Runtime: `<hh:mm:ss>`

---

3. Analysis of Requirements Coverage

<!-- If all requirements are covered sufficently, state "All requirements covered" -->

<For EACH requirement with missing coverage, provide the following structure:>

Requirement `<id>`
Title: `<title>`
Acceptance: `<acceptance>`

Covered Behavior:
`<Describe which part of the accptance is already covered by which test.>`

Missing Behavior:
`<Describe which part of the acceptance is currently uncovered by any test. Add important
↪ information like values to test.>`

---

4. Areas for Improvement
```

```
### Missing Test Coverage (Testable Gaps)
- `<Specific module/function lacking tests that COULD be tested>`
- `<Edge cases not covered>`
- `<Error handling paths not tested (that are reachable)>`

### Untestable Code (Acceptable Gaps)
- `<Code that cannot be tested in current environment with explanation>`
- `<Platform-specific code not executable>`
- `<Defensive checks for impossible states>`

### Test Quality Issues
- `<Tests with unclear assertions>`
- `<Tests missing important edge cases>`
- `<Slow or flaky tests needing refactor>`

---

## 5. Future Work

### Verdict: `<REWORK | FINAL>`

**Justification:**
```text
<Clear explanation of the decision:
- If REWORK: List specific critical testable gaps that must be addressed
- If FINAL: Confirm all tests pass and coverage is adequate given the context and mention
↪  possible improvements>
```
```

C Requirements Document for `simplejson`

```yaml
simplejson_test_requirements:
  scope_and_environment:
    - id: SCOPE-1
      title: Public API exposure
      description: The package must provide the public functions and classes defined in its
       ↪ documentation: dump, dumps, load, loads, JSONEncoder, and JSONDecoder. This ensures
       ↪ compatibility with the Python stdlib json API.
      acceptance: Importing simplejson exposes the documented functions and classes.
    - id: SCOPE-2
      title: Pure-Python functionality
      description: The package includes an optional C extension for performance. However, all
       ↪ functionality must work without the extension to guarantee portability across
       ↪ environments.
      acceptance: All documented behaviors succeed with or without the C extension.

  encoding:
    core_correctness:
      - id: ENC-CORE-1
        title: dumps returns JSON string
        description: Calling dumps(obj) must return a Python str containing valid JSON that
         ↪ conforms to RFC rules. The returned type must never be bytes.
        acceptance: Type is str; round-tripping via loads yields equivalent Python objects.
      - id: ENC-CORE-2
        title: dump writes JSON to file-like
        description: Calling dump(obj, fp) must serialize obj to a text stream by invoking
         ↪ fp.write(str). The output must match the string returned by dumps.
        acceptance: Writing to an in-memory text stream produces correct JSON content.

    options_and_defaults:
      - id: ENC-OPTS-1
        title: ensure_ascii
        description: By default (True), all non-ASCII characters must be escaped as \uXXXX
         ↪ sequences. When False, characters may appear unescaped in the output string.
        acceptance: Unicode character is escaped by default, unescaped with ensure_ascii=False.
      - id: ENC-OPTS-2
        title: skipkeys
        description: If False (default), attempting to serialize a dict with a non-basic key
         ↪ (e.g., tuple, object) raises TypeError. If True, such key-value pairs must be
         ↪ omitted silently.
        acceptance: Tuple key raises TypeError by default; omitted when skipkeys=True.
      - id: ENC-OPTS-3
        title: check_circular
        description: If True (default), encoder must detect self-referential objects and raise
         ↪ an error. If False, circular references will cause recursion depth errors or
         ↪ undefined behavior.
        acceptance: Self-referential list encodes with detection enabled; fails otherwise.
      - id: ENC-OPTS-4
        title: indent formatting
        description: If indent is None (default), output is compact with minimal whitespace. If
         ↪ an integer, that many spaces are used per level. If a string, the string is
         ↪ repeated per level.
```

```
  acceptance: With indent set, output contains newlines and indentation; with None, it
    ↪ does not.
- id: ENC-OPTS-5
  title: separators
  description: Separators define tuple of (item_separator, key_separator). Default is ('
    ↪ ', ': ') if indent is None, else (',', ': '). Custom values must override spacing.
  acceptance: Verify exact output matches chosen separators.
- id: ENC-OPTS-6
  title: sort_keys and item_sort_key
  description: If sort_keys=True, object members must be sorted by key name. item_sort_key
    ↪ provides a callable to determine custom sort order, which overrides sort_keys.
  acceptance: Dict keys are sorted alphabetically or according to custom callable.
- id: ENC-OPTS-7
  title: default function and for_json
  description: If an object is not serializable, the default(obj) callable must be invoked.
    ↪ If for_json=True, objects with a for_json() method must use its return value.
  acceptance: Custom object encodes via default or via its for_json() result.
- id: ENC-OPTS-8
  title: use_decimal for encoding
  description: If True (default), decimal.Decimal objects must serialize as numeric
    ↪ literals with full precision. If False, they are converted to floats.
  acceptance: Decimal('1.1') encodes as "1.1" with use_decimal=True.
- id: ENC-OPTS-9
  title: namedtuple_as_object
  description: If True (default), namedtuple objects must encode as JSON objects using
    ↪ _asdict(). If False, they encode as lists.
  acceptance: Namedtuple serializes as object with field names.
- id: ENC-OPTS-10
  title: tuple_as_array
  description: If True (default), tuple values must serialize as JSON arrays. If False, a
    ↪ TypeError is raised.
  acceptance: (1,2) serializes as [1,2].
- id: ENC-OPTS-11
  title: iterable_as_array
  description: If True, arbitrary objects implementing __iter__ must encode as JSON
    ↪ arrays. If False (default), only list/tuple/known containers are supported.
  acceptance: Custom iterable serializes as array when enabled.
- id: ENC-OPTS-12
  title: bigint_as_string and int_as_string_bitcount
  description: If bigint_as_string=True or int_as_string_bitcount=n, integers with
    ↪ magnitude >= 2**n must encode as strings. This prevents precision loss in
    ↪ JavaScript and similar environments.
  acceptance: Large integers beyond threshold encoded as strings.
- id: ENC-OPTS-13
  title: NaN and Infinity handling
  description: If allow_nan=False (default), encoding float('nan'), float('inf'), or
    ↪ float('-inf') raises ValueError. If allow_nan=True, they encode as
    ↪ NaN/Infinity/-Infinity (non-standard JSON). If ignore_nan=True, such values encode
    ↪ as null.
  acceptance: Validate each behavior mode with explicit test values.
- id: ENC-OPTS-14
  title: JSONEncoderForHTML
  description: Specialized encoder escapes &, <, >, U+2028, and U+2029 for safe embedding
    ↪ in HTML/JavaScript contexts, regardless of ensure_ascii.
```

```
      acceptance: Encoded output contains safe escape sequences for all listed characters.

decoding:
  core_correctness:
    - id: DEC-CORE-1
      title: loads returns Python objects
      description: loads(str) must parse valid JSON text into the correct Python objects
       ↪  (dict, list, int, float, bool, None). Behavior must match the stdlib json module.
      acceptance: Canonical JSON examples decode to expected objects.
    - id: DEC-CORE-2
      title: load consumes entire stream
      description: load(fp) must read until EOF and fail if trailing content remains after a
       ↪  valid JSON document. It does not allow extra characters beyond whitespace.
      acceptance: Input with trailing data raises error.

  options_and_defaults:
    - id: DEC-OPTS-1
      title: object_hook and object_pairs_hook
      description: If provided, object_hook is called with each decoded dict. If
       ↪  object_pairs_hook is provided, it receives ordered pairs and takes precedence.
       ↪  Both enable custom type mapping.
      acceptance: JSON object is transformed into a custom type by the hook.
    - id: DEC-OPTS-2
      title: parse_float, parse_int, parse_constant
     description: These parameters allow replacing the parsing function for floats, ints, and
       ↪  special constants (NaN, Infinity, -Infinity). They must be invoked during parsing.
      acceptance: parse_float=Decimal returns Decimal objects for floats.
    - id: DEC-OPTS-3
      title: use_decimal for decoding
      description: If True, acts as shorthand for parse_float=Decimal. Guarantees lossless
       ↪  parsing of decimal values.
      acceptance: "1.1" decodes to Decimal('1.1') when enabled.
    - id: DEC-OPTS-4
      title: NaN and Infinity decoding
      description: With allow_nan=False (default), NaN/Infinity/-Infinity literals in input
       ↪  must cause failure. With allow_nan=True, they parse into Python float equivalents.
      acceptance: Invalid by default; valid floats if enabled.
    - id: DEC-OPTS-5
      title: strict control character handling
      description: With strict=True (default), control characters in strings must be escaped
       ↪  or an error is raised. If strict=False, unescaped control characters are permitted.
      acceptance: Verify failure under strict=True and acceptance under False.
    - id: DEC-OPTS-6
      title: BOM handling
      description: A leading UTF-8 BOM in input must be ignored during decoding. JSON text
       ↪  must decode successfully even if prefixed by a BOM.
      acceptance: BOM-prefixed input decodes without error.
    - id: DEC-OPTS-7
      title: Oversized integer handling
      description: Extremely large integers (more than ~4300 digits) must raise an error
       ↪  unless a different parse_int function is provided.
      acceptance: Oversized integer fails by default; succeeds with parse_int=Decimal.

compliance_and_interop:
```

```
  - id: COMP-1
    title: Encoding output without BOM
    description: Encoded JSON must not include a BOM. Default encoding should be UTF-8
     ↪  without BOM to ensure interoperability.
    acceptance: Encoded output contains no BOM.
  - id: COMP-2
    title: Top-level primitive values
    description: JSON documents consisting of a single primitive value (number, string,
     ↪  boolean, null) must be accepted.
    acceptance: "42" and "true" decode correctly.
  - id: COMP-3
    title: Repeated object member names
    description: If an object has duplicate member names, the last occurrence overrides
     ↪  earlier ones.
    acceptance: {"a":1,"a":2} decodes to {"a":2}.
  - id: COMP-4
    title: YAML subset compatibility
    description: The default JSON output (with default separators) must form a subset of YAML
     ↪  1.0/1.1/1.2. This ensures compatibility with YAML parsers.
    acceptance: Default separators verified.

error_handling:
  - id: ERR-1
    title: JSONDecodeError attributes
    description: Parsing invalid input must raise simplejson.JSONDecodeError (subclass of
     ↪  ValueError). The exception object must include details: msg, doc, pos, lineno, colno,
     ↪  and optionally end, endlineno, endcolno.
    acceptance: Trigger parse error and verify attributes.
  - id: ERR-2
    title: Unsupported type and NaN errors
    description: Encoding unsupported types without default/for_json must raise TypeError.
     ↪  Encoding NaN/Infinity with allow_nan=False must raise ValueError.
    acceptance: Assert both exceptions raised as specified.

cli:
  - id: CLI-1
    title: Pretty-print tool
    description: Running python -m simplejson.tool with valid JSON input must output
     ↪  formatted JSON to stdout or outfile, adding indentation and line breaks.
    acceptance: Input {"json":"obj"} outputs formatted JSON.
  - id: CLI-2
    title: CLI error reporting
    description: If invalid JSON is provided, the tool must exit with an error message
     ↪  indicating the location of the parse error.
    acceptance: Invalid input prints error message with caret.
  - id: CLI-3
    title: CLI argument defaults
    description: CLI must accept optional infile and outfile arguments. If not provided,
     ↪  stdin and stdout are used by default.
    acceptance: Behavior matches with/without arguments.

non_goals:
  - id: NG-1
    title: Performance guarantees
```

```yaml
  description: No minimum performance levels are required; benchmarks are out of scope.
- id: NG-2
  title: Internal C extension details
  description: The implementation of the optional C extension is not under test; only
  ↪ functional behavior matters.
- id: NG-3
  title: Undocumented JSON extensions
  description: Behaviors not mentioned in documentation (e.g., extra non-standard
  ↪ extensions) are not subject to testing.
```

D Evaluation Results

D.1 Test counts

Trial ID	First Sufficient Test Suite			Last Passing Test Suite		
	Total Tests	**Skipped Tests**	**Execution Times (*s*)**	**Total Tests**	**Skipped Tests**	**Execution Times (*s*)**
0	-	-	-	128	1	7.493
1	93	0	5.193	101	0	5.611
2	27	1	2.000	130	0	7.802
3	-	-	-	104	0	5.264
4	59	1	2.505	124	0	6.541
5	-	-	-	95	4	4.313
6	66	2	2.972	141	1	7.956
7	51	2	2.253	84	1	3.981
8	-	-	-	92	0	4.759
9	-	-	-	126	1	6.926
10	-	-	-	111	2	5.544
11	-	-	-	95	1	4.455
12	-	-	-	108	0	5.417
13	118	2	5.698	139	2	7.298
16	30	1	1.559	76	1	3.648
17	32	0	1.538	91	0	4.674
18	89	0	4.666	110	0	5.931
19	-	-	-	146	0	10.224
20	-	-	-	101	0	4.874
21	46	0	2.116	70	1	3.863
22	97	0	4.682	109	0	5.609
23	-	-	-	68	2	2.800
mean	64	1	3.198	107	1	5.681
median	59	1	2.505	106	0	5.480

Table A.1: Test suite size and execution time of the First Sufficient Test Suites (FSSs) (if avail.) & Last Passing Test Suites (LPSs) for all trials

Trial ID	First Sufficient Test Suite			Last Passing Test Suite		
	Unit Tests	**Integration Tests**	**System Tests**	**Unit Tests**	**Integration Tests**	**System Tests**
0	-	-	-	119	11	0
1	89	4	0	97	4	0
2	24	3	0	124	6	0
3	-	-	-	100	4	0
4	57	2	0	122	2	0
5	-	-	-	90	5	0
6	63	3	0	135	7	0
7	43	8	0	70	14	0
8	-	-	-	86	9	3
9	-	-	-	120	6	0
10	-	-	-	105	7	6
11	-	-	-	86	9	0
12	-	-	-	103	13	0
13	113	5	0	133	7	0
16	27	4	3	72	5	4
17	29	3	0	84	8	0
18	84	5	0	104	6	0
19	-	-	-	136	11	0
20	-	-	-	94	8	0
21	41	5	0	63	8	0
22	94	3	0	106	3	0
23	-	-	-	59	10	0
mean	60	4	0	100	7	1
median	57	4	0	102	7	0

Table A.2: Test type count of the First Sufficient Test Suites (FSSs) (if avail.) & Last Passing Test Suites (LPSs) for all trials

D.2 Coverages

Trial ID	Total Lines	Covered Lines	Line Coverage (%)	Total Branches	Covered Branches	Branch Coverage (%)
1	953	750	78.70	462	348	75.32
2	953	615	64.53	462	268	58.01
4	953	698	73.24	462	309	66.88
6	953	771	80.90	462	319	69.05
7	953	704	73.87	462	320	69.26
13	953	803	84.26	462	346	74.89
16	953	681	71.46	462	291	62.99
17	953	675	70.83	462	299	64.72
18	953	727	76.29	462	332	71.86
21	953	686	71.98	462	306	66.23
22	953	815	85.52	462	356	77.06
mean	953	720	75.60	462	317	68.75
median	953	704	73.87	462	319	69.05

Table A.3: Coverages of the First Sufficient Test Suites (FSSs) for all trials (if avail.)

Trial ID	Total Lines	Covered Lines	Line Coverage (%)	Total Branches	Covered Branches	Branch Coverage (%)
0	953	832	87.30	462	371	80.30
1	953	750	78.70	462	348	75.32
2	953	847	88.88	462	378	81.82
3	953	826	86.67	462	362	78.35
4	953	820	86.04	462	353	76.41
5	953	756	79.33	462	351	75.97
6	953	811	85.10	462	351	75.97
7	953	800	83.95	462	347	75.11
8	953	804	84.37	462	351	75.97
9	953	823	86.36	462	359	77.71
10	953	792	83.11	462	339	73.38
11	953	752	78.91	462	342	74.03
12	953	814	85.41	462	345	74.68
13	953	819	85.94	462	353	76.41
16	953	799	83.84	462	343	74.24
17	953	813	85.31	462	358	77.49
18	953	807	84.68	462	354	76.62
19	953	815	85.52	462	353	76.41
20	953	791	83.00	462	341	73.81
21	953	782	82.06	462	339	73.38
22	953	825	86.57	462	363	78.57
23	953	699	73.35	462	315	68.18
mean	953	798	83.84	462	350	75.91
median	953	809	84.89	462	351	75.97

Table A.4: Coverages of the Last Passing Test Suites (LPSs) for all trials

D.3 Mutation Testing

Trial ID	Total Mutants	Killed Mutants	Mutation Score MS (%)
1	1184	656	55.41
2	1184	100	8.45
4	1184	324	27.36
6	1184	553	46.71
7	1184	337	28.46
13	1184	654	55.24
17	1184	512	43.24
18	1184	671	56.67
21	1184	555	46.88
22	1184	729	61.57
mean	1184	509	43.00
median	1184	554	46.79

Table A.5: Mutation testing results of the First Sufficient Test Suites (FSSs) for all trials (if avail.)[15]

[15]Note, that Trial 16 is not included in this table as `mutmut` failed to execute its FSS.

Trial ID	**Total Mutants**	**Killed Mutants**	**Mutation Score *MS* (%)**
0	1184	746	63.01
1	1184	664	56.08
2	1184	752	63.51
3	1184	728	61.49
4	1184	719	60.73
5	1184	672	56.76
6	1184	683	57.69
7	1184	687	58.02
8	1184	700	59.12
9	1184	725	61.23
10	1184	696	58.78
11	1184	698	58.95
12	1184	717	60.56
13	1184	737	62.25
16	1184	674	56.93
17	1184	720	60.81
18	1184	721	60.9
19	1184	728	61.49
20	1184	675	57.01
21	1184	671	56.67
22	1184	743	62.75
23	1184	605	51.1
mean	1184	703	59.36
median	1184	709	59.84

Table A.6: Mutation Testing Results of the Last Passing Test Suites (LPSs) for all trials

D.4 Requirement counts and Hallucinations

Trial ID	Number of referenced Requirements	Number of referenced & existing Requirements	Number of hallucinated Requirements	Percentage of hallucinated Requirements
1	43	36	7	16.28
2	30	30	0	0
4	48	35	13	27.08
6	44	35	9	20.45
7	37	35	2	5.41
13	47	36	11	23.40
16	37	36	1	2.70
17	34	34	0	0
18	39	36	3	7.69
21	51	34	17	33.33
22	49	36	13	26.53
mean	42	35	7	14.81
median	43	35	7	16.28

Table A.7: Requirement counts of the First Sufficient Test Suites (FSSs) for all trials (if avail.)

Trial ID	Number of referenced Requirements	Number of referenced & existing Requirements	Number of hallucinated Requirements	Percentage of hallucinated Requirements
0	60	36	24	40
1	43	36	7	16.28
2	56	34	22	39.29
3	62	37	25	40.32
4	49	36	13	26.53
5	47	34	13	27.66
6	61	28	33	54.10
7	40	36	4	10
8	49	36	13	26.53
9	56	36	20	35.71
10	44	36	8	18.18
11	36	35	1	2.78
12	39	36	3	7.69
13	51	36	15	29.41
16	37	36	1	2.70
17	39	36	3	7.69
18	45	36	9	20
19	57	36	21	36.84
20	38	35	3	7.89
21	56	35	21	37.5
22	49	36	13	26.53
23	39	36	3	7.69
mean	48	35	13	23.70
median	48	36	13	26.53

Table A.8: Requirement counts of the Last Passing Test Suites (LPSs) for all trials