A Survey on Unit Testing and Large Language Models: Enhancing Software Testing Efficiency

www.surveyx.cn

Abstract

This survey explores the intersection of unit testing and Large Language Models (LLMs) in enhancing software testing efficiency. Unit testing is highlighted as a crucial practice within the software development lifecycle, ensuring code reliability and quality, particularly in safety-critical systems. The emergence of LLMs marks a transformative shift, significantly improving the diversity and quality of unit tests. These models enhance automated unit test generation, address technical challenges, and provide intelligent support for defect detection. Despite their potential, integrating LLMs poses challenges, including the validation and repair of LLM-generated code. The survey delves into automated testing frameworks tailored to specific programming languages, advanced techniques in automated testing, and specialized frameworks for unique testing needs. It examines LLMs' capabilities in test case generation, showcasing their role in improving test coverage and quality. Challenges such as test case reliability, readability, and barriers to unit testing adoption are addressed. Future directions highlight emerging trends, research opportunities, and the integration of LLMs with advanced testing capabilities. The survey underscores the synergy between traditional unit testing practices and the advancements offered by LLMs, vital for enhancing testing efficiency and meeting modern software development demands.

1 Introduction

1.1 Importance of Unit Testing in Software Development

Unit testing is a cornerstone of the software development lifecycle, crucial for enhancing software quality by countering the tendency of developers to prioritize feature implementation over test creation [1]. It ensures that the code adheres to specified requirements and passes both overt and covert tests, thereby ensuring reliability across software components, including ONNX operators. This practice is essential in safety-critical systems like embedded software, where software quality assurance is imperative [2].

Unit testing is instrumental in identifying and mitigating various bugs, particularly Single Statement Bugs (SSBs), which can adversely affect functionality and lead to significant losses. By promoting efficient code production with fewer defects, unit testing enhances overall software quality [3]. Additionally, it upholds code integrity and diminishes the likelihood of bugs, thereby reinforcing the software development process [4].

1.2 Emergence of Large Language Models

The emergence of Large Language Models (LLMs) signifies a transformative shift in software testing, markedly improving the diversity and quality of unit tests. These models address the critical issue of inadequate diversity and volume in unit test datasets, leading to more robust testing practices [5]. LLMs, exemplified by those used in Methods2Test, utilize extensive supervised datasets to enhance automated unit test generation, thereby facilitating more effective model training and evaluation [1].

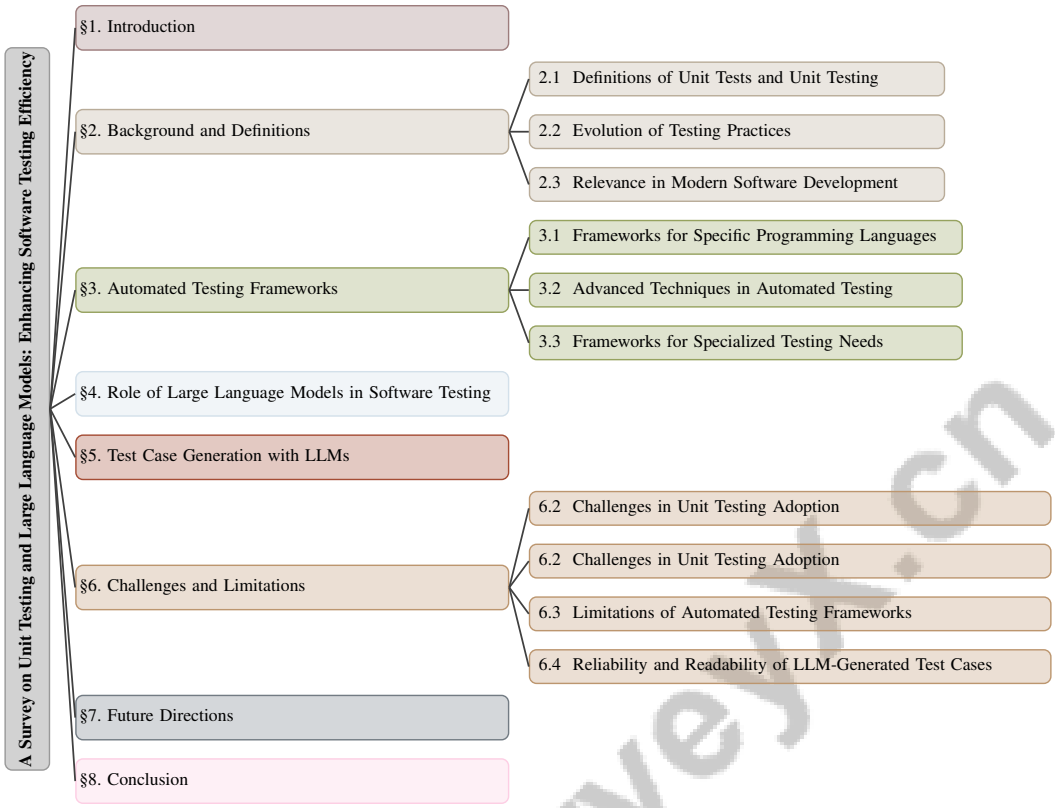


Figure 1: chapter structure

The integration of LLMs into software testing workflows, as illustrated by the LLM Programming Workflow (LPW), highlights their potential to improve code generation accuracy through plan verification [6]. The exceptional performance of pre-trained language models in generating function signatures and associated unit tests underscores their role in automating and streamlining testing processes [7].

Furthermore, LLMs tackle technical challenges such as non-fully qualified names and syntax errors through innovative solutions like the Partial Code Reuse Chain (PCR-Chain), which breaks down compilation tasks into manageable steps [8]. The introduction of models like AUGER, which combines defect detection with error triggering, exemplifies LLMs' potential to enhance testing efficiency by providing intelligent support in identifying and resolving software defects [9].

Despite these advancements, the integration of LLMs in software testing presents challenges. Developers must navigate the complexities of validating and repairing code generated by LLMs, as highlighted in studies on developer interactions with tools like GitHub Copilot [10]. These challenges emphasize the necessity of understanding code provenance and ensuring that generated code aligns with expected behaviors.

1.3 Structure of the Survey

This survey offers a comprehensive exploration of the intersection between unit testing and the application of Large Language Models (LLMs) in enhancing software testing efficiency. It commences with an introduction that emphasizes the significance of unit testing in software development and discusses the role of automated testing frameworks alongside the emergence of LLMs. The subsequent background and definitions section provides foundational knowledge by defining key concepts such as unit tests, automated testing, and LLMs, while also tracing the evolution of testing practices and their contemporary relevance.

Following this, the survey examines automated testing frameworks, detailing their features, advantages, and limitations, with a focus on frameworks tailored to specific programming languages,

advanced techniques, and specialized testing needs. The role of LLMs in software testing is explored, highlighting their capabilities in test case generation, provision of intelligent suggestions, and the challenges encountered.

The survey further investigates test case generation using LLMs, presenting case studies and examples that illustrate their effectiveness, as well as techniques for enhancing test case quality and coverage. Challenges and limitations are discussed in a dedicated section, addressing barriers to unit testing adoption, constraints of automated testing frameworks, and concerns regarding the reliability and readability of LLM-generated test cases.

Finally, the paper outlines future directions, emphasizing emerging trends and research opportunities, the integration of LLMs with advanced testing capabilities, and their potential support for diverse programming languages and environments. The conclusion underscores the critical role of unit testing and LLM application in improving testing efficiency, highlighting recent research that demonstrates LLMs' effectiveness in generating and refining unit tests. These studies reveal that while LLMs can automate test case creation, enhancing coverage and reliability, further refinement in test quality is necessary, particularly in reducing common test smells. The findings highlight the potential of LLMs, such as Meta's TestGen-LLM, to not only generate new tests but also improve existing ones, thereby streamlining the testing process and contributing to superior software quality [11, 12, 13, 14].The following sections are organized as shown in Figure 1.

2 Background and Definitions

2.1 Definitions of Unit Tests and Unit Testing

Unit tests are automated checks of individual software components, ensuring they operate correctly and fulfill specified requirements [4]. They are crucial in the software development lifecycle for early defect detection, thereby improving reliability and quality. These tests support test-driven development by facilitating targeted test case creation [9]. Beyond functionality validation, unit testing is vital for accurate code translation across languages, adhering to standards like IEEE 1788-2015 for interval arithmetic libraries [15]. Their scarcity poses challenges in training code synthesis models [7]. Unit testing also mitigates Single Statement Bugs (SSBs), significantly impacting software functionality [16]. Automated tools enhance developer efficiency by suggesting tests that increase confidence in defect detection methods [9].

As a cornerstone of modern software engineering, unit tests verify software correctness and reliability, forming the base of the testing pyramid. In agile environments, collaboration between testers and developers promotes comprehensive test case design and continuous improvement. Automated tools that generate meaningful assertions are crucial for software correctness assessment, facilitating early defect identification and improving software quality and maintainability [14, 17, 18, 19, 20]. As methodologies evolve, the scope and efficacy of unit testing are expected to expand, reinforcing their significance in the development lifecycle.

2.2 Evolution of Testing Practices

Software testing practices have evolved to meet the complexities of modern systems. Initially manual and error-prone, the shift to automated frameworks marked a milestone, enhancing reliability and efficiency while reducing development cycles [21]. This shift accommodated object-oriented paradigms, introducing complexities like encapsulation and polymorphism [22]. Testing practices now include patterns like Simple Test and Assertion Message for structured behavior verification [23]. Mutation operators tailored to new language features, such as C++11/14, demonstrate ongoing refinement to meet evolving requirements [24].

Advancements in monitoring and instrumentation, such as distributed tracing, improve issue diagnosis in complex systems [25]. In education, the limitations of automated grading tools for diverse programming skills have prompted reevaluation of testing practices, especially for languages like C++ [26]. Test-Driven Development (TDD) exemplifies the iterative nature of practices, emphasizing pre-implementation testing [27]. Multithreaded software testing challenges, such as concurrency issues, necessitate advanced techniques [28]. Consistent strategies are needed for testing interval arithmetic libraries due to varying implementations [29].

The dynamic evolution of testing practices underscores the need for innovative tools grounded in historical insights to maintain efficacy in the software industry. This enhances unit testing quality, traditionally a developer domain, while emphasizing the collaborative role of testers in agile environments, contributing to quality improvement initiatives. Assertion messages in test cases are vital for validating behavior, with practitioner insights highlighting their role in troubleshooting and understanding failures. Focusing on these aspects fosters cooperation and continuous learning, leading to improved software quality and testing methodologies [18, 14].

2.3 Relevance in Modern Software Development

Unit testing remains essential in modern software development for ensuring quality and reliability. The integration of Parameterized Unit Tests (PUTs) expands input space coverage, crucial for managing modern application complexities [30]. Methodologies like AUT-XSS address vulnerabilities such as cross-site scripting, enhancing testing robustness [31]. The Basic Block Coverage (BBC) approach exemplifies search-based methodologies for improving branch coverage [32].

High-quality datasets for automated test generation are vital for enhancing quality and fault detection [1], particularly in safety-critical embedded software [2]. The correlation between coverage and defect reduction underscores comprehensive testing's importance [20]. Challenges persist in validating and repairing code generated by Large Language Models (LLMs), requiring developers to ensure accuracy [10].

LLMs impact testing methodologies significantly, generating test cases from bug reports and addressing complex bugs [33]. However, the ROBUSTAPI benchmark reveals API misuse in LLM-generated code, highlighting the need for oversight [34]. LLMs synthesize API usage examples from unit tests, addressing insufficient client code challenges [35].

Unit testing's relevance is further underscored by engaging practices like gamification to maintain high testing coverage and quality [36]. Introducing unit testing to high school students highlights its foundational role in software engineering education [37]. Tools like AUGER, integrating defect detection and test generation, improve testing efficiency, underscoring current relevance [9]. The lack of a standard framework for C++ complicates testing, emphasizing effective unit testing's necessity in modern environments [4].

3 Automated Testing Frameworks

Category	Feature	Method
Frameworks for Specific Programming Languages	Compile and Code Verification	LLM4PR[38], CTMTF[4]
	Automation and Management	SJ[39], AFPS[40]
	Logic-Specific Testing	ASP-WIDE[41]
Advanced Techniques in Automated Testing	Solution Validation	LPW[6]
	Test Input Generation	N/A[42], FuzzAug[5]
	Path Exploration Techniques	SU[2]
	Reinforcement Learning Methods	AUTD-RL[7]
Frameworks for Specialized Testing Needs	Visual and Interactive Testing	C[43]
	AI-Driven Test Automation	CMT[44]
	Real-World Data Utilization	JTG[45]

Table 1: The table presents a comprehensive overview of various automated testing frameworks, categorized into three primary areas: specific programming languages, advanced techniques, and specialized testing needs. Each category details the features and methods utilized by different frameworks, highlighting their unique contributions to enhancing software testing efficiency and reliability. This structured summary facilitates an understanding of the diverse strategies employed to address the challenges associated with modern software development.

Automated testing frameworks play a pivotal role in modern software development by enhancing software quality and reliability. Table 1 provides a detailed overview of the automated testing frameworks discussed in this section, categorizing them by their focus areas and outlining the key features and methods they employ. Additionally, Table 2 offers a structured comparison of various automated testing frameworks, emphasizing their focus areas, techniques, and compatibility, as discussed in this section. As illustrated in Figure 2, these frameworks can be categorized into three main areas: specific programming languages, advanced techniques, and specialized testing needs. This figure highlights the tools and strategies used in each category, demonstrating how they address

unique challenges and characteristics inherent in various programming paradigms, thereby optimizing testing processes. This examination of frameworks tailored for specific programming languages will further elucidate their contributions to improving software quality and reliability.

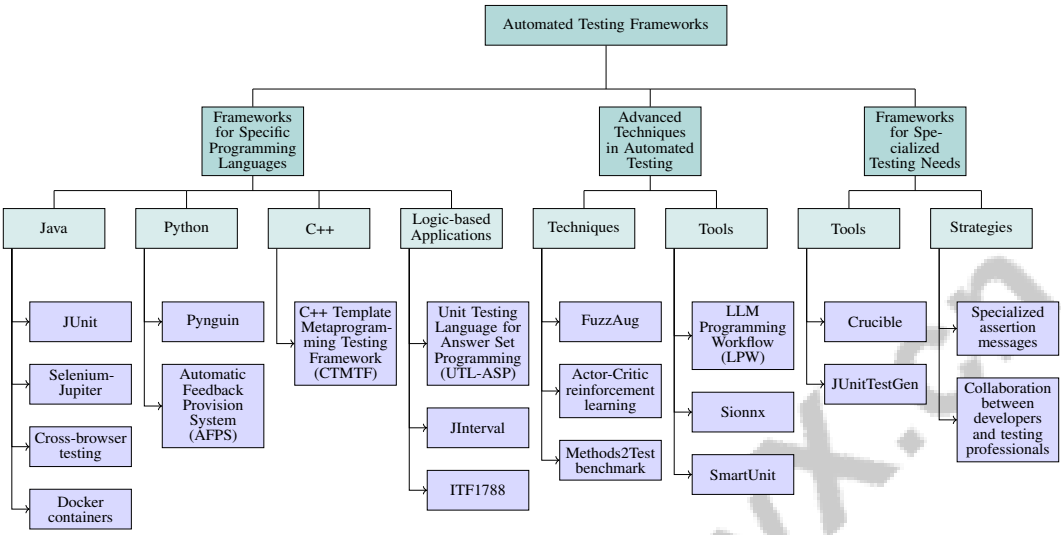


Figure 2: This figure illustrates the categorization of automated testing frameworks into three main areas: specific programming languages, advanced techniques, and specialized testing needs. It highlights the tools and strategies used in each category to enhance software quality and reliability.

3.1 Frameworks for Specific Programming Languages

Frameworks designed for specific programming languages provide tools that significantly enhance unit testing efficiency. In Java, JUnit, often paired with Selenium-Jupiter, facilitates comprehensive testing by automating browser driver management and supporting cross-browser testing through Docker containers [39]. The analysis of 7,824 Single Statement Bugs (SSBs) in Maven-based Java projects on GitHub underscores the importance of these frameworks in benchmarking and improving software quality [16].

For Python, Pynguin employs search-based techniques to generate regression tests with high code coverage, while the Automatic Feedback Provision System (AFPS) enhances educational outcomes by providing automatic feedback on programming assignments [40]. The C++ Template Metaprogramming Testing Framework (CTMTF) exemplifies a compile-time approach, eliminating the need for external tools and streamlining testing for template metaprograms [4]. Logic-based applications benefit from the Unit Testing Language for Answer Set Programming (UTL-ASP), which facilitates test case creation and execution against ASP programs, while frameworks like JInterval and ITF1788 support testing for interval arithmetic libraries [46, 29].

These frameworks not only bolster software development robustness but also address challenges posed by diverse programming paradigms. By employing specialized tools and methodologies, including Test-Driven Development (TDD) and agile collaboration, developers can create comprehensive test cases and foster continuous improvement, leading to higher-quality software that meets user needs [18, 47].

Figure 3 illustrates the application and utility of automated testing frameworks across different programming languages. It showcases scenarios such as a list of symbols fundamental to code logic, a Python snippet using graph theory in a simulation, and a "Converter" class for financial computations, emphasizing the importance of automated testing in enhancing code robustness [38, 41, 48].

3.2 Advanced Techniques in Automated Testing

Advanced techniques in automated testing frameworks significantly enhance testing efficiency by addressing modern development complexities. FuzzAug, for instance, employs fuzzing to generate

$$
\begin{aligned}
\langle Type \rangle &::= \texttt{bool} \mid \texttt{nat} \mid \texttt{int} \mid \texttt{float} \mid \texttt{char} \mid \texttt{array}\langle Type \rangle \\
\langle Expr \rangle &::= \langle Number \rangle \mid \langle Name \rangle \mid \texttt{true} \mid \texttt{false} \mid \langle Variable \rangle \\
&\mid \langle Expr \rangle == \langle Expr \rangle \mid \langle Expr \rangle < \langle Expr \rangle \mid \langle Expr \rangle <= \langle Expr \rangle \\
&\mid \langle Expr \rangle > \langle Expr \rangle \mid \langle Expr \rangle >= \langle Expr \rangle \mid \langle Expr \rangle \; != \langle Expr \rangle \\
&\mid \langle Expr \rangle \text{ and } \langle Expr \rangle \mid \langle Expr \rangle \text{ or } \langle Expr \rangle \mid \text{not} \langle Expr \rangle \\
&\mid \langle Expr \rangle + \langle Expr \rangle \mid \langle Expr \rangle - \langle Expr \rangle \\
&\mid \langle Expr \rangle \cdot \langle Expr \rangle \mid \langle Expr \rangle \; / \; \langle Expr \rangle \\
&\mid \langle Variable \rangle [\langle Expr \rangle] \\
&\mid \langle Variable \rangle [\langle Expr \rangle : \langle Expr \rangle] \\
\langle Prog \rangle &::= \texttt{pass} \mid \langle Variable \rangle = \langle Expr \rangle \mid \langle Prog \rangle ; \; \langle Prog \rangle \\
&\mid \texttt{while } (\langle Expr \rangle) : \langle Prog \rangle \\
&\mid \texttt{if } (\langle Expr \rangle) : \langle Prog \rangle \texttt{ else } \langle Prog \rangle \\
&\mid \texttt{assert} \langle Expr \rangle \\
&\mid \texttt{def } \langle Name \rangle \; (\langle Name \rangle : \langle Type \rangle) * : \langle Prog \rangle
\end{aligned}
$$

(a) The image shows a list of symbols and expressions in a programming language.[38]

```
%** @block(name="hamCycle") **%

%** @rule(name="r1", block="hamCycle") **%
inCycle(X,Y) | outCycle(X,Y) :- arc(X,Y).

%** @rule(name="r2", block="hamCycle") **%
reached(X) :- start(X).

%** @rule(name="r3", block="hamCycle") **%
reached(Y) :- reached(X), inCycle(X,Y).

%** @rule(name="r4", block="hamCycle") **%
:- inCycle(X,Y), inCycle(X,Z), Y<>Z.

%** @rule(name="r5", block="hamCycle") **%
:- inCycle(X,Y), inCycle(Z,Y), X<>Z.

%** @rule(name="r6", block="hamCycle") **%
:- node(X), not reached(X).

%** @test(name = "checkProperty",
        scope = { "hamCycle" },
        input = "node(1). node(2). node(3). node(4). arc(1,2). arc(1,4).
            arc(2,4). arc(3,1). arc(4,3). start(1)."
        assert = { @constraintForAll("':-node(X), #count{Y:inCycle(X,Y)}=0.") }
    )
**%
```

(b) A Python code snippet demonstrating the use of graph theory concepts in a simulation environment[41]

```
"""Exercise 1
--Keywords--
currency
class
function
parameters
dictionary
exchange
--Problem statement--
Write a class called Converter that is initialized with a dictionary of exchange rates for
-- currencies
against the USD, e.g. {"USD": 1, "EUR": 0.9, "GBP": 0.75}. The class should have a method
-- called
convert, which takes in three parameters: from_currency, to_currency, and amount. The
-- function should
return the given amount converted from the first currency (first parameter) to the second
-- currency
(second parameter) using the exchange rate dictionary given in the class constructor.

As an example, the code
converter = Converter({"USD": 1, "EUR": 0.9, "GBP": 0.75})
in_euros = converter.convert("USD", "EUR", 100)
print(in_euros)
should print out 90.0
--Sample solution--
class Converter():
  def __init__(self, exchange_rates):
    self.exchange_rates = exchange_rates

  def convert(self, from_currency, to_currency, amount):
    amount_in_usd = amount / self.exchange_rates[from_currency]
    return amount_in_usd * self.exchange_rates[to_currency]
--Test--
class TestConverter(unittest.TestCase):
  def test_converter(self):
    converter = Converter({"USD": 1, "EUR": 0.8})
    self.assertEqual(converter.convert("USD", "EUR", 100), 80)

  def test_converter2(self):
    converter = Converter({"USD": 1, "EUR": 0.9, "GBP": 0.75, "SEK": 9.71})
    self.assertEqual(converter.convert("USD", "USD", 100), 100)
    self.assertEqual(converter.convert("USD", "EUR", 100), 90)
    self.assertEqual(converter.convert("GBP", "EUR", 10), 12)
    self.assertEqual(converter.convert("EUR", "GBP", 10), 8.333333333333332)
```

(c) A Python code snippet demonstrating the creation of a class called "Converter" with methods to convert amounts between different currencies using a dictionary of exchange rates.[48]

Figure 3: Examples of Frameworks for Specific Programming Languages

diverse, high-quality test inputs, bolstering unit test robustness [5]. Integrating automatic data generation with an Actor-Critic reinforcement learning framework, as demonstrated by Gorinski et al., optimizes test scenarios, thereby enhancing reliability [7]. The Methods2Test benchmark provides a dataset reflecting real-world tests, allowing machine learning models to learn from high-quality examples and improve unit test generation [1].

The LLM Programming Workflow (LPW) employs solution generation and verification through visible tests, ensuring generated solutions meet requirements before integration [6]. Similarly, Sionnx automates unit test generation for ONNX operators based on predefined specifications, streamlining machine learning model testing [42]. SmartUnit uses dynamic symbolic execution to enhance coverage metrics and comprehensively evaluate software behavior, exploring execution paths to identify defects not apparent through traditional methods [2].

These advanced techniques contribute to the evolution of automated testing frameworks, offering innovative solutions to modern software development challenges. By employing methodologies like TDD and integrating assertion messages, developers can enhance testing effectiveness and efficiency, fostering a culture of continuous improvement [14, 17, 47, 18, 19].

Figure 4 illustrates the hierarchical categorization of advanced techniques in automated testing, highlighting three main areas: data augmentation, test generation, and assertion and verification. Each category includes specific methods that contribute to enhancing testing efficiency and reliability. The first example in the figure depicts a test case processing workflow streamlined by large language models (LLMs). The second showcases the BART Transformer Model Architecture's role in generating accurate test assertions, while the third outlines systematic test case generation from bug reports, demonstrating the integration of advanced algorithms in software testing [13, 17, 33].

3.3 Frameworks for Specialized Testing Needs

Frameworks tailored for specialized testing needs offer solutions to unique software testing challenges. Crucible provides a graphical interface for creating AUnit test cases for Alloy models, simplifying test case creation by translating visual representations into textual formats [43]. In mobile application development, JUnitTestGen generates unit test cases for Android APIs by mining real-world API usage, ensuring tests reflect actual usage patterns [45]. This enhances Android application robustness while reducing manual effort in creating comprehensive test suites.

These specialized frameworks represent innovative strategies to meet diverse software testing needs. By focusing on targeted domains and leveraging specialized knowledge, these tools empower developers with resources that improve software quality and reliability, ensuring adaptation to unique application requirements. This tailored approach promotes effective unit testing through specialized assertion messages and fosters collaboration between developers and testing professionals,

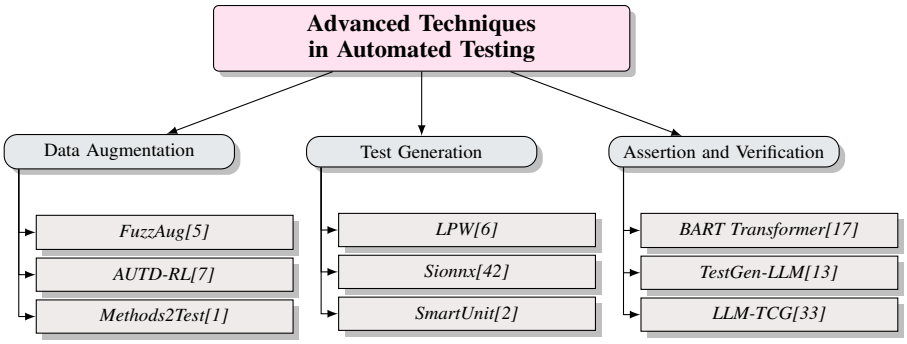
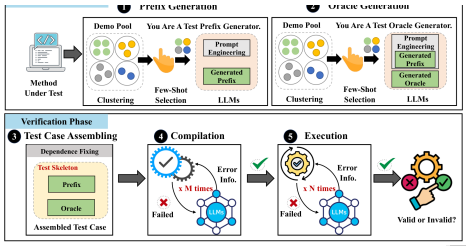
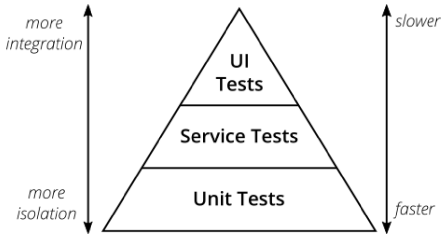


Figure 4: This figure illustrates the hierarchical categorization of advanced techniques in automated testing, highlighting three main areas: data augmentation, test generation, and assertion and verification. Each category includes specific methods that contribute to enhancing testing efficiency and reliability.

enhancing software quality. Educational insights underscore the importance of addressing common pitfalls in unit testing practices, reinforcing the value of domain-specific tools in improving software development outcomes [14, 49, 50, 18, 51].



(a) Test Case Assembly and Verification Process[44]



(b) The Pyramid of Testing Levels[52]

Figure 5: Examples of Frameworks for Specialized Testing Needs

As shown in Figure 5, automated testing frameworks streamline testing processes and ensure software reliability. The first example outlines a methodical approach to assembling and verifying test cases, segmented into Prefix Generation, Oracle Generation, and Verification phases. The second example categorizes tests into Unit Tests, Service Tests, and UI Tests, emphasizing the balance needed for robust software validation. These frameworks exemplify tailored strategies in automated testing to effectively address specific testing requirements [44, 52].

Feature	Frameworks for Specific Programming Languages	Advanced Techniques in Automated Testing	Frameworks for Specialized Testing Needs
Testing Focus	Unit Testing	Efficiency Enhancement	Specialized Challenges
Technique Used	Search-based Techniques	Fuzzing, Reinforcement Learning	Graphical Interface, Api Mining
Programming Compatibility	Java, Python, C++	Machine Learning Models	Mobile Applications

Table 2: This table presents a comparative analysis of automated testing frameworks, categorized by their testing focus, techniques used, and programming compatibility. The frameworks are divided into three main areas: those specific to programming languages, advanced techniques in automated testing, and frameworks for specialized testing needs. This classification highlights the diverse approaches and tools employed to enhance software testing efficiency and address specific challenges within different programming paradigms.

4 Role of Large Language Models in Software Testing

The integration of Large Language Models (LLMs) into software testing marks a pivotal advancement, addressing traditional challenges and enhancing test methodologies to ensure software reliability and quality. This section explores LLM capabilities in improving test case generation, leveraging

their deep understanding of code semantics to enhance both coverage and quality, thereby reshaping software testing practices.

4.1 Capabilities of LLMs in Test Case Generation

LLMs have revolutionized test case generation by employing a nuanced understanding of code semantics and syntax, resulting in enhanced coverage and quality. They automate meaningful API usage examples, as demonstrated by the API Usage Example Synthesis from Unit Tests (AUES-UT) method, which synthesizes examples from unit tests to ensure comprehensive real-world application alignment [35]. The ATHENATEST model exemplifies LLM capabilities by generating realistic, readable test cases from focal methods, significantly improving coverage through a sequence-to-sequence transformer model that learns from real-world examples [6].

Techniques like FuzzAug enhance training datasets for neural test generation, enabling LLMs to create semantically meaningful test cases [5]. The combination of automatic data generation with reinforcement learning contributes to large datasets of function signatures and unit tests, bolstering testing reliability [7]. Moreover, LLMs assist in generating test cases for code translations, leveraging unit tests to validate and improve coverage [15].

The Partial Code Reuse Chain (PCR-Chain) method showcases LLMs' ability to resolve non-fully qualified names and correct syntax errors in partial code through a structured AI chain approach [8]. The AUGER method employs attention mechanisms to guide LLMs in focusing on defective statements during unit test generation, significantly enhancing efficiency [9]. SmartUnit's dynamic symbolic execution engine exemplifies LLM capabilities in maximizing statement, branch, and MC/DC coverage [2]. This approach enables comprehensive evaluation of software behavior, identifying potential defects not easily detected by traditional methods.

As illustrated in Figure 6, the transformative capabilities of LLMs in test case generation focus on automated API usage, enhanced coverage, and testing efficiency. Key methods and models are highlighted, showcasing their contributions to improving test case generation processes. The integration of these advanced techniques significantly advances coverage, improves quality, and ensures software application reliability. As testing models evolve, their integration is expected to enhance robustness and efficiency, fostering collaboration between developers and testing professionals to create comprehensive test cases and promote a culture of quality improvement. Tools leveraging LLMs are anticipated to streamline accurate assert statement generation and enhance test case understandability, ultimately elevating software product quality through thorough validation and effective troubleshooting throughout the development lifecycle [14, 17, 53, 11, 18].

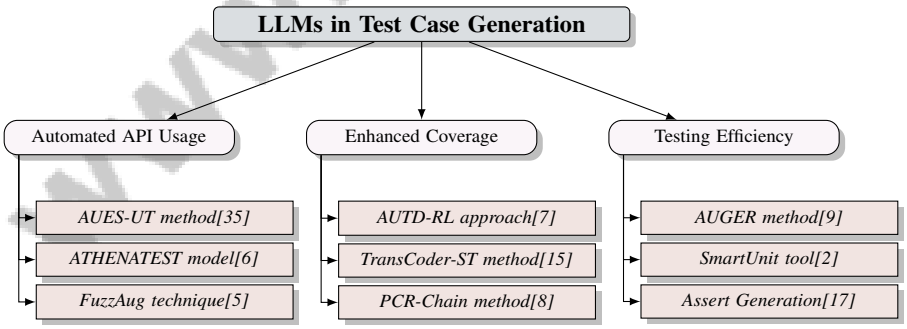


Figure 6: This figure illustrates the transformative capabilities of large language models (LLMs) in test case generation, focusing on automated API usage, enhanced coverage, and testing efficiency. Key methods and models are highlighted, showcasing their contributions to improving test case generation processes.

4.2 Intelligent Suggestions and Automation

LLMs have significantly reshaped software testing by providing intelligent suggestions and automating various testing processes. Leveraging their advanced understanding of code semantics, these models offer real-time support that enhances testing efficiency. The RLTF framework exemplifies this capability by generating training data in real-time and utilizing multi-granularity feedback from unit

tests to guide the model in producing higher-quality code [54]. This iterative feedback loop aligns generated code with developer expectations, ensuring quality standards are met.

The ATHENATEST model demonstrates LLM effectiveness in generating test cases that align closely with developer expectations. By learning from high-quality examples, ATHENATEST produces tests that meet functional requirements and adhere to software testing best practices [55]. This capability reduces manual effort in test creation and enhances overall testing efficiency.

Moreover, the modular design of the Partial Code Reuse Chain (PCR-Chain) facilitates effective interaction with LLMs, improving code compilation reliability and resolving common issues like non-fully qualified names and syntax errors [8]. This structured approach enables LLMs to provide intelligent suggestions for code correction, streamlining development and minimizing errors.

In educational settings, AI tools powered by LLMs offer timely support by generating hints that assist students in identifying and correcting code errors. This application enhances the learning experience by providing immediate feedback and guidance [50], fostering a deeper understanding of programming concepts and improving student outcomes.

The integration of LLMs in software testing processes significantly enhances automation by streamlining test case generation and improving existing test suites. For instance, Meta's TestGen-LLM tool has demonstrated that LLMs can refine human-written tests, achieving a 75% success rate in generating valid test cases and increasing coverage in 25% of instances during practical deployments. Comprehensive evaluations indicate that while LLM-generated tests hold promise for automating unit test generation, improvements in test quality, particularly in reducing common test smells, remain necessary. This body of research highlights LLMs' potential to transform testing workflows through intelligent suggestions and automation, paving the way for more efficient and reliable software development practices [11, 12, 13]. By leveraging advanced models, developers can achieve higher efficiency and reliability in software testing, ultimately leading to improved software quality and reduced development time.

4.3 Challenges and Limitations of LLMs in Software Testing

The integration of LLMs in software testing presents various challenges and limitations that can affect their effectiveness and reliability. A primary concern is the potential for LLM-generated tests to deviate from established best practices, resulting in ineffective testing outcomes. This issue is compounded by the generation of malformed tests, necessitating additional validation and correction to ensure testing process reliability. LLMs often struggle to generate meaningful assertions, as existing methods may yield incomplete or overly simplistic assertions that fail to capture the complexity needed to detect software faults [19].

Additionally, LLMs face challenges in accurately classifying correct assertions amid buggy code, indicating a reliance on actual implementation for effective test generation. Designing effective tests that satisfy multiple requirements without excessive test proliferation remains a significant hurdle. Existing methods often struggle with this, particularly concerning mutation operators, where tools may fail to cover specific faults introduced by new programming features, leading to undetected faults [24].

LLMs also encounter limitations in static analysis, which can lead to false negatives and inadequate identification of dependencies in complex test scenarios. This necessitates refining methodologies to improve test coverage accuracy and completeness. The reliance on existing test cases may create gaps in coverage, as benchmarks may not encompass all possible edge cases, especially in specialized domains like interval arithmetic [29].

From an optimization perspective, combining multiple criteria in the testing process can result in local optima traps and reduced search effectiveness. This complexity is exacerbated by challenges in existing monitoring methods, which often fail to capture internal function calls necessary for effective software visualization, primarily focusing on distributed tracing [25].

Moreover, the LLM Programming Workflow (LPW) faces limitations, including LLM reasoning capacity and substantial token consumption for generating plans and verifications, which can hinder application in resource-constrained environments [6]. Developers may overlook the LLM origin of code, complicating validation and repair strategies, and complicating LLM integration into existing workflows [10].

Inherent limitations in proposed frameworks, such as the inability to test certain runtime aspects (e.g., database interactions and multithreaded program behavior), further complicate the landscape [4]. Addressing these challenges is crucial to realizing LLMs' full potential in advancing software testing practices, ensuring software systems' reliability and quality. Ongoing research and development are essential to overcome these limitations and enhance LLM efficacy in software testing.

5 Test Case Generation with LLMs

The integration of Large Language Models (LLMs) into test case generation represents a significant advancement in computational techniques and software testing methodologies. As the demand for efficient testing processes increases, LLMs have emerged as pivotal tools, automating and enhancing various aspects of test case generation. This section presents case studies and examples that highlight the effectiveness of LLMs in this domain.

5.1 Case Studies and Examples

Benchmark	Size	Domain	Task Format	Metric
LLM-TOG[56]	1,000	Software Testing	Test Oracle Classification	Accuracy, Mutation Score
TestGen-LLM[13]	1,979	Software Testing	Test Case Generation	Coverage Improvement, Reliability
MTPB[57]	115	Programming	Multi-turn Program Synthesis	Pass Rate
LLM4TS[12]	216,300	Software Testing	Unit Test Generation	Correctness, Readability
KC-HEA[50]	872	Programming Education	Code Improvement	Overlap with Top-3 Missing KCs
TESTPILOT[58]	1,684	Software Testing	Unit Test Generation	Statement Coverage, Branch Coverage
JUnit-LLM[59]	411	Unit Testing	Unit Test Generation	Line Coverage, Branch Coverage
ITL[60]	14,000	Interval Arithmetic	Unit Testing	Accuracy, Compliance

Table 3: This table presents a comprehensive overview of various benchmarks utilized in the application of large language models (LLMs) for software testing and programming tasks. It details the benchmark names, sizes, domains, task formats, and evaluation metrics, illustrating the diversity and scope of LLM-driven test case generation and program synthesis methodologies. The included benchmarks highlight the effectiveness and challenges faced by LLMs in improving software testing processes.

The application of LLMs in test case generation has been extensively explored through various case studies, demonstrating their effectiveness in improving software testing processes. For instance, the Reinforcement Learning from Static Quality Metrics (RLSQM) framework has shown enhancements in unit test quality generated by LLMs, optimizing test generation through reinforcement learning [61]. Despite producing test oracles that align with program behavior, LLMs exhibit less than 50% accuracy in classification tasks, indicating a need for better alignment with developer expectations [56].

An analysis of over 9,000 GitHub projects and 188,154 test methods with single assert statements revealed the potential of LLMs to generate meaningful assertions, though challenges in achieving complete fault detection remain [19]. The InspectIT Ocelot agent demonstrated significant improvements in capturing internal and distributed traces, enhancing software behavior visualization [25]. In regression testing, LLMs achieved a 97% reduction in re-executed tests, particularly beneficial for safety-critical software development [62].

The AUGER model exhibited notable improvements in defect detection and unit test generation, with enhanced F1-score and Precision, effectively directing LLMs to focus on defective statements [9]. Comprehensive studies of LLMs generating unit tests for Java classes have revealed improvements in test correctness and coverage, while identifying areas for refinement, such as reducing common test smells. Meta's TestGen-LLM tool achieved a 75% success rate in generating correct test cases during test-a-thons. Method slicing techniques have significantly improved coverage by breaking complex methods into manageable components for LLMs, surpassing traditional test generation techniques [63, 12, 13]. Table 3 provides a detailed overview of the benchmarks used in evaluating the performance of large language models in software testing and programming domains.

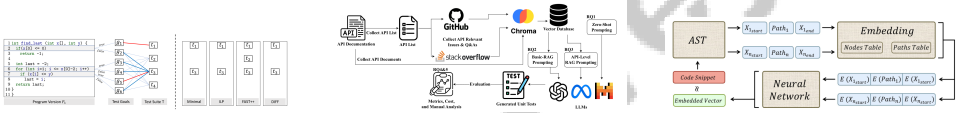
5.2 Techniques for Test Case Enhancement

Enhancing test case quality and coverage is crucial for robust software testing, with LLMs playing a significant role through innovative techniques. Prompt optimization offers actionable insights for improving code generation, enabling LLMs to produce higher-quality test cases with better coverage [64]. SELF-DEBUGGING allows LLMs to iteratively refine outputs through generation, explanation, and feedback, ensuring functionally accurate test cases that align with expected behaviors [65].

Integrating unit testing and model validation techniques enhances test case quality and coverage by ensuring both software functionality and model reliability [66]. Advanced methods like context-aware ranked tree automata and pattern-matching recursion schemes improve precision by accurately identifying success or failure conditions [67]. The success typing method facilitates error detection by mapping input types to crash conditions, enhancing test case quality [68].

Execution feedback-based rejection sampling, used in AUTOIF, enhances instruction quality by retaining only the most relevant test cases [69]. Techniques like Basic Block Coverage (BBC) prioritize tests based on their coverage of critical code parts, maximizing testing impact in complex systems [32]. Friendly composition and hiding operations enhance integration testing by minimizing ambiguous states and ensuring component compatibility [70].

These techniques demonstrate diverse strategies for utilizing LLMs to enhance test case quality and coverage. Studies indicate that LLMs can effectively generate unit tests with improved coverage through methods such as method slicing and prompt engineering. Tools like Meta's TestGen-LLM have been successfully implemented in industrial settings, yielding measurable improvements in test reliability and coverage, contributing to robust software testing practices [13, 71, 12, 63, 72].



(a) Program Version P1 and Test Goals and Test Suite T[73]

(b) API APIQA: A System for APIQA with API-Level RAG Prompting[74]

(c) Code Snippet Embedding with AST and Neural Network[75]

Figure 7: Examples of Techniques for Test Case Enhancement

As illustrated in Figure 7, leveraging LLMs in test case generation and enhancement offers innovative techniques to optimize software testing processes. The examples depicted illustrate various methodologies employed to enhance test cases. The first example, "Program Version P1 and Test Goals and Test Suite T," showcases a program version with a function identifying the last element in an array meeting specific criteria, evaluated through a test suite represented as a directed acyclic graph (DAG), emphasizing structured evaluation of test goals. The second example, "API APIQA: A System for APIQA with API-Level RAG Prompting," presents a flowchart detailing the API question-answering process that integrates data from GitHub and Stack Overflow to train models, demonstrating the potential of API-level relevance attention in refining test cases. Lastly, "Code Snippet Embedding with AST and Neural Network" highlights embedding code snippets using abstract syntax trees (AST) and neural networks, indicating a sophisticated method for understanding and enhancing code through machine learning. Collectively, these examples underscore the diverse applications of LLMs in advancing test case generation and the continuous improvement of software testing methodologies [73, 74, 75].

6 Challenges and Limitations

The challenges associated with unit testing are multifaceted and encompass both technical and human factors that impede its widespread adoption in software development. To gain a comprehensive understanding of the barriers faced in unit testing practices, it is crucial to investigate the specific challenges that students encounter, such as the difficulty in identifying test smells in their code due to a lack of experience, and the impact of abstract concepts on their engagement and motivation in learning software testing. Insights from recent studies highlight that students often struggle with bad unit testing practices, which can lead to the introduction of problematic code in their test suites. Moreover,

the implementation of automated marking in online unit testing challenges has been shown to enhance student engagement and understanding, providing timely feedback that aids in the identification of mistakes and misconceptions. [51, 76]. This exploration will illuminate the complexities involved and highlight the critical areas that require attention for successful implementation.

6.1 Challenges in Unit Testing Adoption

The adoption of unit testing practices within software development is often met with various challenges that can impede progress and effectiveness. As illustrated in Figure 9, these challenges can be categorized into three main areas: technical complexities, human factors, and tooling/environmental issues. The figure highlights key barriers associated with each category and their implications for the successful implementation of unit testing.

Technical complexities often arise from the intricacies involved in creating effective unit tests that adequately cover the codebase. Additionally, human factors, such as resistance to change or lack of training, can significantly hinder the adoption of these practices. Finally, tooling and environmental issues, including inadequate testing frameworks or integration difficulties, further complicate the landscape of unit testing. Understanding these challenges is crucial for organizations aiming to enhance their software quality through robust testing methodologies.

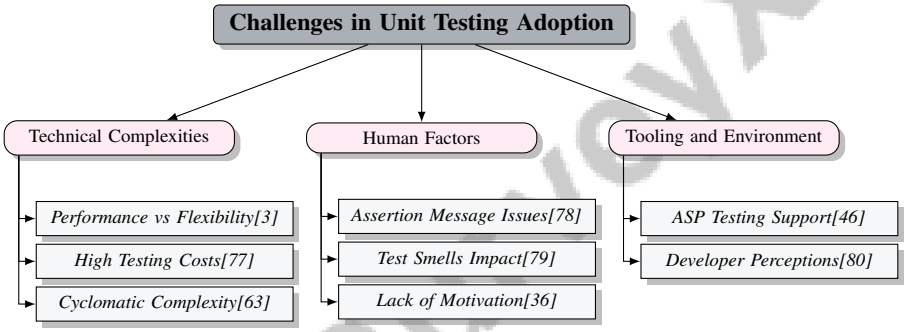


Figure 8: This figure illustrates the primary challenges in adopting unit testing practices, categorized into technical complexities, human factors, and tooling/environmental issues, highlighting key barriers and their implications in software development.

6.2 Challenges in Unit Testing Adoption

The adoption of unit testing practices in software development is fraught with challenges that span technical complexities and human factors. One notable barrier is the struggle to achieve performance levels comparable to static languages while retaining the flexibility inherent in dynamic languages [3]. This dichotomy complicates the integration of unit testing into development workflows, particularly in environments that prioritize adaptability and rapid iteration.

The inherent complexity of modern software systems also poses significant obstacles. High testing costs and the absence of structured test code often deter the widespread implementation of unit testing practices [77]. This complexity is magnified in scenarios involving methods with high cyclomatic complexity, which complicates the generation of comprehensive tests and reduces overall test coverage [63]. The co-evolution of test and production code can lead to design erosion, further complicating the maintenance of effective unit testing over time [81].

Developers frequently encounter challenges related to assertion messages, which are crucial for debugging and maintaining software quality. The underutilization of informative assertion messages hampers troubleshooting efforts and can degrade software quality [78]. Crafting effective assertion messages that remain pertinent throughout the software lifecycle, while ensuring their clarity and diagnostic value, remains a persistent challenge [14].

The presence of test smells, such as assertion roulette, is another impediment to the adoption of unit testing practices. A lack of understanding of these smells and their impact on debugging can lead to inefficiencies and compromise software quality [79]. Additionally, the increased cognitive load

on students when debugging, coupled with inadequate feedback from automated marking systems, presents further barriers to effective unit testing education and practice.

Moreover, a lack of motivation among developers and limited engagement from testing professionals can hinder the integration of unit testing into the development workflow [36]. Limited experience among students in recognizing detrimental test smells also poses a barrier, affecting the ability to adopt and implement effective unit testing practices.

Finally, the analysis of developers' perceptions of test smells across different projects reveals variations that indicate barriers to adopting effective unit testing practices [80]. Inadequate support for testing in existing ASP development environments presents another obstacle to effective unit testing [46]. Addressing these barriers is crucial for realizing the full potential of unit testing in enhancing software quality and reliability.

Figure 9 illustrates the primary challenges in adopting unit testing practices, categorized into technical complexities, human factors, and tooling/environmental issues, highlighting key barriers and their implications in software development.

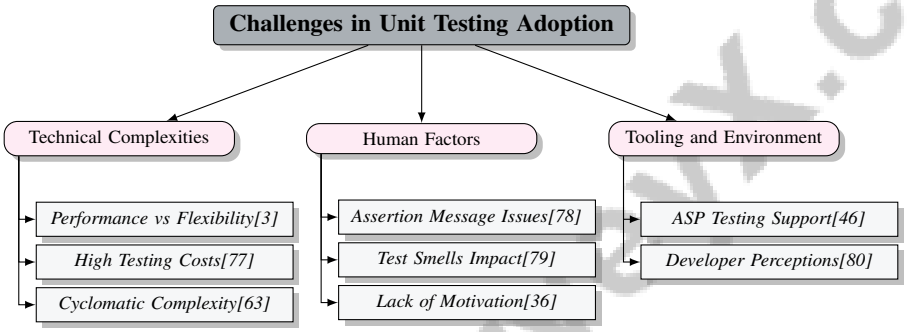


Figure 9: This figure illustrates the primary challenges in adopting unit testing practices, categorized into technical complexities, human factors, and tooling/environmental issues, highlighting key barriers and their implications in software development.

6.3 Limitations of Automated Testing Frameworks

Automated testing frameworks, while pivotal in enhancing the efficiency of software testing, are not without their limitations and constraints. A significant challenge these frameworks face is their reliance on manual test case generation, which can be both time-consuming and result in inadequate test coverage [2]. This dependency not only increases the workload for developers but also limits the scalability and effectiveness of the testing process. Moreover, the complexities involved in integrating various programming paradigms can introduce usability challenges, further complicating the testing process [3].

Traditional automated testing frameworks often rely on external tools, creating additional overhead and complexity in the testing process [4]. This reliance can be cumbersome, particularly in environments that require seamless integration and minimal disruption to existing workflows. Additionally, the use of handwritten test cases, as seen in frameworks for verifying ONNX operator compliance, underscores the inefficiencies inherent in current methodologies [9]. Such approaches not only demand significant manual effort but also struggle to achieve comprehensive test coverage.

The AUGER model addresses some of these limitations by offering a more integrated and efficient approach to unit test generation, thereby enhancing the overall testing process [9]. However, despite these advancements, the complexities of adapting automated testing frameworks to diverse programming environments remain a significant challenge. The need for frameworks that can seamlessly integrate with multiple programming paradigms without introducing additional complexity is critical for advancing the field.

6.4 Reliability and Readability of LLM-Generated Test Cases

The reliability and readability of test cases generated by Large Language Models (LLMs) are critical in determining their efficacy in software testing. Despite the potential of LLMs, concerns

persist regarding the prevalence of test smells and poor design in the generated test cases, which can undermine their reliability and readability . These issues are further exacerbated by the high prevalence of API misuse and limitations in current evaluation methods, which highlight the need for improved robustness and reliability in LLM-generated code [34].

The analysis of test messages indicates that those composed of string literals tend to be more readable than those made up of identifiers, impacting the overall reliability of LLM-generated test cases [78]. Additionally, while LLMs like ChatGPT show promise in generating unit tests, they do not yet outperform traditional tools like EvoSuite in terms of code coverage and bug detection, suggesting room for improvement in test case reliability [82].

Furthermore, the accuracy of assertions in LLM-generated test cases remains a significant concern, as a substantial percentage of assertions may be incorrect, necessitating more precise assertion generation [83]. The challenges faced by LLMs in handling less common programming languages and providing comprehensive evaluation metrics further underscore the need for enhanced methodologies to ensure the reliability and readability of generated test cases [84].

The study of developer behaviors with LLM-generated code emphasizes the importance of code provenance awareness, which can improve validation performance and reduce cognitive workload, suggesting that informed developers engage more effectively with LLM-generated test cases [10]. However, challenges remain, particularly in specific edge cases or under certain prompt conditions, which can affect both the reliability and readability of the test cases [8].

7 Future Directions

7.1 Emerging Trends and Research Opportunities

The integration of Large Language Models (LLMs) in software testing is opening new avenues for research and development. The creation of extensive datasets like Methods2Test offers valuable resources for training LLMs, addressing prior dataset limitations and improving unit test generation capabilities [1]. These advancements are essential for enhancing software quality through more effective testing methodologies.

Tools such as Sionnx facilitate automated test generation for AI model compliance, improving reliability by ensuring adherence to standards [42]. Similarly, SmartUnit enhances unit testing in embedded software, demonstrating the refinement of testing strategies for specific domains [2]. Future research may focus on enhancing LLM reasoning and integrating alternative solution representations to improve code generation across diverse programming contexts [6]. Exploring multimodal capabilities in LLMs for code generation and validation also presents promising research directions [10].

Expanding frameworks like PCR-Chain to support additional programming languages and scenarios could improve LLM effectiveness in resolving syntax errors and non-fully qualified names [8]. Further, enhancing models such as AUGER and validating their effectiveness through human studies will be critical [9]. Investigating testing effectiveness across various bug types, programming languages, and closed-source projects could inform the development of more comprehensive methodologies [16]. Enhancing the extensibility of systems like Ciao could lead to new trends in adaptable testing frameworks [3].

Developing specialized specification languages is crucial for improving the precision and expressiveness of test specifications, leading to more effective testing processes [4]. These emerging trends highlight the transformative potential of LLMs in advancing software testing methodologies, promising robust and efficient practices to meet modern software development demands.

7.2 Integration of LLMs with Advanced Testing Capabilities

Integrating Large Language Models (LLMs) with advanced testing capabilities offers a promising approach to enhancing software testing processes. Future research could focus on augmenting frameworks with sophisticated methodologies that leverage LLM strengths. For instance, incorporating FuzzAug, a data augmentation technique using fuzzing to generate valid test inputs, could significantly enhance test diversity and quality [5]. This integration would create robust testing environments capable of addressing diverse input scenarios, ensuring comprehensive coverage.

Combining LLMs with frameworks like WiP Unit Testing Framework could enhance testing capabilities by integrating advanced features, improving adaptability and effectiveness in various scenarios [85]. This advancement would streamline testing processes, reducing manual effort and allowing developers to focus on complex tasks.

The potential for LLMs to improve test case generation using advanced techniques such as reinforcement learning and symbolic execution is promising. Methodologies like prompt engineering with LLMs such as GPT and Mistral have shown significant improvements in accuracy and efficiency, facilitating comprehensive test suites that address issues like test smell detection and coverage gaps. LLMs can generate executable test cases from bug reports, crucial for fault localization and automated repair, streamlining testing processes. Tools like Meta's TestGen-LLM have enhanced test cases, achieving improvements in reliability and coverage during industrial deployments [86, 12, 13, 33]. This integration of LLMs with advanced capabilities promises to revolutionize software testing, equipping developers with powerful tools for ensuring software reliability and robustness.

7.3 Support for Diverse Programming Languages and Environments

The ability of Large Language Models (LLMs) to support a broader range of programming languages and environments is a promising research area, driven by the need for versatile testing frameworks. As the variety of languages and environments expands, the demand for LLMs capable of integrating with multiple languages to deliver robust solutions across platforms increases. Studies highlight the potential of LLMs like GPT and Mistral in automating unit test generation and improving test cases, though challenges remain in ensuring quality and readability. Tools like Meta's TestGen-LLM have shown promise in enhancing human-written tests, achieving improvements in coverage and reliability during industrial applications [87, 12, 13].

Future research will likely focus on enhancing frameworks to accommodate additional languages and testing levels, refining LLMs to understand and generate code in multiple languages, facilitating effective cross-language testing [73]. By supporting a broader range of languages, LLMs can foster inclusive environments that meet diverse projects' needs.

Integrating LLMs with advanced capabilities across different environments will be crucial for this expansion. Leveraging LLM strengths, developers can implement advanced methodologies addressing challenges associated with diverse paradigms and environments. Studies illustrate LLM potential, particularly for complex Java projects, where techniques like method slicing simplify input analysis and enhance coverage of conditions and branches. This approach increases the correctness and understandability of generated tests while addressing common pitfalls like test smells, leading to more robust and efficient processes [63, 12]. This integration will enhance frameworks' adaptability and improve overall testing efficiency and effectiveness.

8 Conclusion

This survey highlights the integral role of unit testing and Large Language Models (LLMs) in advancing software testing methodologies and enhancing software quality. Unit testing is established as a fundamental practice that fosters improved testing practices and developer awareness, especially within deep learning environments. The adoption of structured testing methodologies, such as the Arrange-Act-Assert (AAA) framework, is emphasized for its clarity and maintainability, with a significant number of test cases adhering to this structure.

Innovative tools, including TestART and CTGEN, mark significant progress in unit test generation, achieving high pass rates and coverage metrics, thereby reducing manual effort and enhancing testing efficiency. Additionally, tools like AssertConvert contribute to maintainability and traceability by generating English representations for JUnit assertions.

LLMs are acknowledged as transformative in software testing, significantly enhancing test generation and bug-fixing capabilities. Their integration into testing processes is exemplified by frameworks such as Pynguin, which address the challenges of dynamically typed languages and improve testing efficiency. The HITS approach further demonstrates the superiority of LLM-based test generation methods over traditional techniques.

The survey underscores the necessity for a comprehensive testing framework in digitally transformed organizations, where software quality is increasingly influenced by data, pipelines, and models. Notably, the Interleaving framework's ability to deterministically reproduce multithreaded bugs represents a substantial advancement over traditional methods.

In educational contexts, understanding test smells is crucial for enhancing software engineering education, equipping students with the skills needed to develop high-quality test cases. This survey underscores the pivotal contributions of unit testing and LLMs in advancing testing methodologies, improving software quality, and meeting the evolving demands of modern software development.

References

[1] Michele Tufano, Shao Kun Deng, Neel Sundaresan, and Alexey Svyatkovskiy. Methods2test: A dataset of focal methods mapped to test cases, 2022.

[2] Chengyu Zhang, Yichen Yan, Hanru Zhou, Yinbo Yao, Ke Wu, Ting Su, Weikai Miao, and Geguang Pu. Smartunit: Empirical evaluations for automated unit testing of embedded software in industry, 2018.

[3] M. V. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J. F. Morales, and G. Puebla. An overview of ciao and its design philosophy, 2011.

[4] Norbert Pataki. Testing by c++ template metaprograms, 2010.

[5] Yifeng He, Jicheng Wang, Yuyang Rong, and Hao Chen. Data augmentation by fuzzing for neural test generation, 2024.

[6] Chao Lei, Yanchuan Chang, Nir Lipovetzky, and Krista A. Ehinger. Planning-driven programming: A large language model programming workflow, 2025.

[7] Philip John Gorinski, Matthieu Zimmer, Gerasimos Lampouras, Derrick Goh Xin Deik, and Ignacio Iacobacci. Automatic unit test data generation and actor-critic reinforcement learning for code synthesis, 2023.

[8] Qing Huang, Jiahui Zhu, Zhenchang Xing, Huan Jin, Changjing Wang, and Xiwei Xu. A chain of ai-based solutions for resolving fqns and fixing syntax errors in partial code, 2023.

[9] Xin Yin, Chao Ni, Xiaodan Xu, and Xiaohu Yang. What you see is what you get: Attention-based self-guided automatic unit test generation, 2024.

[10] Ningzhi Tang, Meng Chen, Zheng Ning, Aakash Bansal, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. A study on developer behaviors for validating and repairing llm-generated code using eye tracking and ide actions, 2024.

[11] Andrea Lops, Fedelucio Narducci, Azzurra Ragone, Michelantonio Trizio, and Claudio Bartolini. A system for automated unit test generation using large language models and assessment of generated test suites, 2024.

[12] Wendkûuni C. Ouédraogo, Kader Kaboré, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F. Bissyandé. Large-scale, independent and comprehensive study of the power of llms for test case generation, 2024.

[13] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at meta, 2024.

[14] Anthony Peruma, Taryn Takebayashi, Rocky Huang, Joseph Carmelo Averion, Veronica Hodapp, Christian D. Newman, and Mohamed Wiem Mkaouer. On the rationale and use of assertion messages in test code: Insights from software practitioners, 2024.

[15] Baptiste Roziere, Jie M. Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation, 2022.

[16] Habibur Rahman and Saqib Ameen. How is testing related to single statement bugs?, 2024.

[17] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. Generating accurate assert statements for unit test cases using pretrained transformers, 2020.

[18] Lucas Neves, Oscar Campos, Robson Santos, Italo Santos, Cleyton Magalhaes, and Ronnie de Souza Santos. Elevating software quality in agile environments: The role of testing professionals in unit testing, 2024.

[19] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases, 2020.

[20] Lucas Gren and Vard Antinyan. On the relation between unit testing and code quality, 2019.

[21] Hafsa Cheddadi, Saad Motahhir, and Abdelaziz El Ghzizal. Google test/google mock to verify critical embedded software, 2022.

[22] Martin Levesque. A metamodel of unit testing for object-oriented programming languages, 2009.

[23] Danielle Gonzalez, Joanna C. S. Santos, Andrew Popovich, Mehdi Mirakhorli, and Mei Nagappan. A large-scale study on the usage of testing patterns that address maintainability attributes (patterns for ease of modification, diagnoses, and comprehension), 2017.

[24] Ali Parsai, Serge Demeyer, and Seph De Busser. C++11/14 mutation operators based on common fault patterns, 2020.

[25] Malte Hansen and Wilhelm Hasselbring. Instrumentation of software systems with opentelemetry for software visualization, 2024.

[26] Vaishnavi Bhargava, Rajat Ghosh, and Debojyoti Dutta. Cpp-ut-bench: Can llms write complex unit tests in c++?, 2024.

[27] Amjed Tahir, Steve Counsell, and Stephen G. MacDonell. An empirical study into the relationship between class features and test smells, 2021.

[28] Evgeny Vainer and Amiram Yehudai. Taming the concurrency: Controlling concurrent behavior while testing multithreaded software, 2014.

[29] Nathalie Revol, Luis Benet, Luca Ferranti, and Sergei Zhilin. Testing interval arithmetic libraries, including their ieee-1788 compliance, 2022.

[30] Deepika Tiwari, Yogya Gamage, Martin Monperrus, and Benoit Baudry. Proze: Generating parameterized unit tests informed by runtime data, 2024.

[31] Mahmoud Mohammadi, Bill Chu, and Heather Richter Lipford. Detecting cross-site scripting vulnerabilities through automated unit testing, 2018.

[32] Pouria Derakhshanfar, Xavier Devroey, and Andy Zaidman. Basic block coverage for search-based unit testing and crash reproduction, 2022.

[33] Laura Plein, Wendkûuni C. Ouédraogo, Jacques Klein, and Tegawendé F. Bissyandé. Automatic generation of test cases based on bug reports: a feasibility study with large language models, 2023.

[34] Li Zhong and Zilong Wang. Can chatgpt replace stackoverflow? a study on robustness and reliability of large language model code generation, 2024.

[35] Mohammad Ghafari, Konstantin Rubinov, and Mohammad Mehdi Pourhashem K. Mining unit test cases to synthesize api usage examples, 2022.

[36] Philipp Straubinger, Tommaso Fulcini, Gordon Fraser, and Marco Torchiano. Intelligame in action: An experience report on gamifying javascript unit tests, 2024.

[37] Joseph Latessa, Aadi Huria, and Deepak Raju. Introducing high school students to version control, continuous integration, and quality assurance, 2023.

[38] Yufan Cai, Zhe Hou, Xiaokun Luan, David Miguel Sanan Baena, Yun Lin, Jun Sun, and Jin Song Dong. Towards large language model aided program refinement, 2024.

[39] Boni García, Carlos Delgado Kloos, Carlos Alario-Hoyos, and Mario Munoz-Organero. Selenium-jupiter: A junit 5 extension for selenium webdriver, 2024.

[40] Hans Fangohr, Neil O'Brien, Anil Prabhakar, and Arti Kashyap. Teaching python programming with automatic assessment and feedback provision, 2015.

[41] Giovanni Amendola, Tobias Berei, Giuseppe Mazzotta, and Francesco Ricca. Unit testing in asp revisited: Language and test-driven development environment, 2024.

[42] Xinli Cai, Peng Zhou, Shuhan Ding, Guoyang Chen, and Weifeng Zhang. Sionnx: Automatic unit test generator for onnx conformance, 2019.

[43] Adam G. Emerson and Allison Sullivan. Crucible: Graphical test cases for alloy models, 2023.

[44] Chao Ni, Xiaoya Wang, Liushan Chen, Dehai Zhao, Zhengong Cai, Shaohua Wang, and Xiaohu Yang. Casmodatest: A cascaded and model-agnostic self-directed framework for unit test generation, 2024.

[45] Xiaoyu Sun, Xiao Chen, Yanjie Zhao, Pei Liu, John Grundy, and Li Li. Mining android api usage to generate unit test cases for pinpointing compatibility issues, 2022.

[46] Onofrio Febbraro, Nicola Leone, Kristian Reale, and Francesco Ricca. Unit testing in aspide, 2011.

[47] Davide Fucci, Hakan Erdogmus, Burak Turhan, Markku Oivo, and Natalia Juristo. A dissection of the test-driven development process: Does it really matter to test-first or to test-last?, 2016.

[48] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. Automatic generation of programming exercises and code explanations using large language models, 2022.

[49] Zhichao Zhou, Yuming Zhou, Chunrong Fang, Zhenyu Chen, Xiapu Luo, Jingzhu He, and Yutian Tang. Coverage goal selector for combining multiple criteria in search-based unit test generation, 2024.

[50] Laryn Qi, J. D. Zamfirescu-Pereira, Taehan Kim, Björn Hartmann, John DeNero, and Narges Norouzi. A knowledge-component-based methodology for evaluating ai assistants, 2024.

[51] Anthony Peruma, Eman Abdullah AlOmar, Wajdi Aljedaani, Christian D. Newman, and Mohamed Wiem Mkaouer. Insights from the field: Exploring students' perspectives on bad unit testing practices, 2024.

[52] Nicole Radziwill and Graham Freeman. Reframing the test pyramid for digitally transformed organizations, 2020.

[53] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. Leveraging large language models for enhancing the understandability of generated unit tests, 2024.

[54] Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. Rltf: Reinforcement learning from unit test feedback, 2023.

[55] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context, 2021.

[56] Michael Konstantinou, Renzo Degiovanni, and Mike Papadakis. Do llms generate test oracles that capture the actual or the expected program behaviour?, 2024.

[57] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[58] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation, 2023.

[59] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. Using large language models to generate junit tests: An empirical study, 2024.

[60] Luis Benet, Luca Ferranti, and Nathalie Revol. A framework to test interval arithmetic libraries and their ieee 1788-2015 compliance, 2023.

[61] Benjamin Steenhoek, Michele Tufano, Neel Sundaresan, and Alexey Svyatkovskiy. Reinforcement learning from automatic feedback for high-quality unit test generation, 2025.

[62] Andrew V. Jones. Addressing the regression test problem with change impact analysis for ada, 2016.

[63] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. Hits: High-coverage llm-based unit test generation via method slicing, 2024.

[64] Zhenlan Ji, Pingchuan Ma, Zongjie Li, and Shuai Wang. Benchmarking and explaining large language model-based code generation: A causality-centric approach, 2023.

[65] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

[66] Gopal P. Sarma, Travis W. Jacobs, Mark D. Watts, Vahid Ghayoomi, Richard C. Gerkin, and Stephen D. Larson. Unit testing, model validation, and biological simulation, 2017.

[67] Robert Jakob and Peter Thiemann. Towards tree automata-based success types, 2013.

[68] Robert Jakob and Peter Thiemann. A falsification view of success typing, 2015.

[69] Guanting Dong, Keming Lu, Chengpeng Li, Tingyu Xia, Bowen Yu, Chang Zhou, and Jingren Zhou. Self-play with execution feedback: Improving instruction-following capabilities of large language models, 2024.

[70] Przemyslaw Daca, Thomas A. Henzinger, Willibald Krenn, and Dejan Nickovic. Compositional specifications for ioco testing, 2019.

[71] Weichao Xu, Huaxin Pei, Jingxuan Yang, Yuchen Shi, Yi Zhang, and Qianchuan Zhao. Exploring critical testing scenarios for decision-making policies: An llm approach, 2024.

[72] Wendkûuni C. Ouédraogo, Yinghua Li, Kader Kaboré, Xunzhu Tang, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F. Bissyandé. Test smells in llm-generated unit tests, 2024.

[73] Sebastian Ruland and Malte Lochau. On the interaction between test-suite reduction and regression-test selection strategies, 2022.

[74] Jiho Shin, Reem Aleithan, Hadi Hemmati, and Song Wang. Retrieval-augmented test generation: How far are we?, 2024.

[75] Mosab Rezaei, Hamed Alhoori, and Mona Rahimi. Test case recommendations with distributed representation of code syntactic features, 2023.

[76] Chakkrit Tantithamthavorn and Norman Chen. Unit testing challenges with automated marking, 2023.

[77] Amjed Tahir and Stephen G. MacDonell. Combining dynamic analysis and visualization to explore the distribution of unit test suites, 2021.

[78] Taryn Takebayashi, Anthony Peruma, Mohamed Wiem Mkaouer, and Christian D. Newman. An exploratory study on the usage and readability of messages within assertion methods of test cases, 2023.

[79] Wajdi Aljedaani, Mohamed Wiem Mkaouer, Anthony Peruma, and Stephanie Ludi. Do the test smells assertion roulette and eager test impact students' troubleshooting and debugging capabilities?, 2023.

[80] Denivan Campos, Larissa Rocha, and Ivan Machado. Developers perception on the severity of test smells: an empirical study, 2021.

[81] Bart Van Rompaey and Serge Demeyer. Exploring the composition of unit test suites, 2007.

[82] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. Chatgpt vs sbst: A comparative assessment of unit test suite generation, 2023.

[83] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. Unit test generation using generative ai : A comparative performance analysis of autogeneration tools, 2024.

[84] Liguo Chen, Qi Guo, Hongrui Jia, Zhengran Zeng, Xin Wang, Yijiang Xu, Jian Wu, Yidong Wang, Qing Gao, Jindong Wang, Wei Ye, and Shikun Zhang. A survey on evaluating large language models in code generation tasks, 2024.

[85] Ponkoj Chandra Shill, David Feil-Seifer, Jiullian-Lee Vargas Ruiz, and Rui Wu. Wip: A unit testing framework for self-guided personalized online robotics learning, 2024.

[86] Rabimba Karanjai, Aftab Hussain, Md Rafiqul Islam Rabin, Lei Xu, Weidong Shi, and Mohammad Amin Alipour. Harnessing the power of llms: Automating unit test generation for high-performance computing, 2024.

[87] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 2024.

Disclaimer:

SurveyX is an AI-powered system designed to automate the generation of surveys. While it aims to produce high-quality, coherent, and comprehensive surveys with accurate citations, the final output is derived from the AI's synthesis of pre-processed materials, which may contain limitations or inaccuracies. As such, the generated content should not be used for academic publication or formal submissions and must be independently reviewed and verified. The developers of SurveyX do not assume responsibility for any errors or consequences arising from the use of the generated surveys.