Rapid Literature Review of Reinforcement Learning and Large Language Model Techniques for Software Engineering Testing and Bug Detection

Daniel de la Calle
Department of Computer Science
The University of Alabama
Tuscaloosa, AL, USA
Email: dgdelacalle@crimson.ua.edu

Md Nazmul Hoque
Department of Computer Science
The University of Alabama
Tuscaloosa, AL, USA
Email: mhoque4@crimson.ua.edu

Ryan Adolfs
Department of Computer Science
The University of Alabama
Tuscaloosa, AL, USA
Email: rmadolfs@crimson.ua.edu

***Abstract*—As the world increasingly relies on software for its smooth functioning, testing of the ever-changing algorithms has become increasingly more difficult as the dimensions of the data, state spaces, and internal logic increases in complexity. Thus, the development of efficient, yet adaptable software testing techniques has become a common priority in ensuring the reliability of the application. This literature review synthesizes recent advancements in Reinforcement Learning (RL) and Large Language Models (LLMs) for software testing and bug detection. RL offers adaptive, exploratory testing, excelling in dynamic environments like continuous integration, autonomous driving, and cybersecurity. RL's ability to prioritize tests and uncover subtle faults enhances reliability in complex systems. Conversely, LLMs, such as GPT-3 and Llama-2, automate test case generation, enhance test oracle accuracy, and improve bug detection. A comparative analysis reveals RL's superiority in interactive, state-rich scenarios, alongside LLMs' strength in code-centric, predictable tasks. Industrial adoption is growing, though challenges like functional inaccuracies, computational costs, and integration complexities persist. This review underscores the complementary roles of LLMs and RL, advocating for hybrid approaches to maximize their impact on intelligent, scalable software quality assurance.**

***Index Terms*—Software Testing Methods, Reinforcement Learning, Large Language Models, Test Automation, Test Case Generation, Test Coverage, Test Oracle Generation, Bug Detection, Fuzz Testing, Mutation Testing, Vulnerability Detection,**

I. Introduction

Software testing is a fundamental pillar of the software development lifecycle, ensuring that applications meet their intended functionality, reliability, and quality standards. It serves as a critical mechanism for identifying and resolving defects, enhancing user satisfaction, and mitigating the risks of system failures that could lead to financial losses or eroded trust. Despite its importance, traditional software testing methodologies face significant challenges that hinder their efficiency and effectiveness, particularly as software systems grow in scale and complexity.

Traditional testing approaches are often labor-intensive and resource-heavy. Manual testing, for instance, requires testers to design and execute test cases, a process that can be prone to human error and inconsistencies, especially in large-scale systems with intricate component interactions. The financial burden of hiring skilled testers, maintaining test infrastructure, and updating test suites as software evolves can be substantial. Moreover, traditional methods frequently fail to achieve comprehensive coverage, missing edge cases or rare scenarios that could lead to critical failures. Scalability is another concern, as these methods struggle to keep pace with the rapid development cycles and complexity of modern software applications. These limitations highlight the urgent need for innovative solutions to streamline and enhance the testing process.

Testing complex software systems, particularly those embedded in dynamic environments, poses considerable challenges for traditional manual and heuristic-based methodologies. Manual testing often demands significant human effort and domain expertise to devise appropriate test scenarios and interpret outcomes. Furthermore, these manual methods become increasingly infeasible when systems exhibit intricate state interactions, evolving behaviors, or massive action spaces.

Recently, Reinforcement Learning (RL) has emerged as a powerful, adaptive, and automated alternative to manual testing, systematically exploring vast state spaces and strategically prioritizing testing actions to efficiently detect faults in software. Unlike manual approaches, RL leverages learning algorithms that dynamically adjust strategies based on past interactions, significantly enhancing both the depth and coverage of tests executed. This adaptive learning capability is especially critical in continuous integration (CI) contexts, autonomous driving systems (ADS), gaming, and other interactive software systems, where testing resources must be utilized optimally due to constraints in computation, time, and cost.

Moreover, RL testing methods address many critical limitations inherent in traditional, manual, or heuristic-based techniques. For instance, manual test case generation and prioritization suffer from human biases and oversights, potentially missing critical scenarios, especially when testing complex, non-deterministic systems. In contrast, RL automates test scenario generation, intelligently prioritizing test cases based on

learned metrics, past execution data, and the system's evolving state space. By applying RL to testing contexts, practitioners have significantly improved bug detection capabilities and overall testing efficiency. For example, in continuous integration environments, RL consistently outperforms heuristic-based methods by more accurately prioritizing tests according to historical failure data, code-change impact analysis, and predictive metrics, thereby reducing regression testing durations and costs without compromising software reliability.

Critically, RL testing methods are not merely automated replacements for manual efforts; they fundamentally transform the testing paradigm. While manual testing emphasizes deterministic, pre-defined tests based on developer expectations and domain knowledge, RL emphasizes exploration, learning, and adaptive experimentation within the software-under-test's operational environment. This shift in paradigm fosters the discovery of subtle, latent defects which traditional methods might overlook due to limited exploration. Furthermore, RL's inherent adaptability allows it to remain effective even when the system under test evolves over time, an essential trait in agile, rapidly iterating software development processes. Consequently, RL testing methods significantly enhance software testing outcomes across various domains by offering a scalable, adaptable, automated, and intelligence-driven testing methodology.

Following the advent of RL testing methods, the community has utilized Large Language Models (LLMs) for unit test generation. As this showed promise, there became a growing interest in automating various aspects of software testing using LLMs (see Fig 1), such as GPT-3 and Codex. Since they have demonstrated remarkable capabilities in understanding and generating code, it makes them very suitable for tasks like bug detection and automating entire testing workflows [1].

For instance, Rehan et al. [2] showcased how LLMs can be fine-tuned to generate unit tests for Java methods with high coverage and reduced redundancy, while Wang et al. [1] surveyed the broader landscape of LLMs in software testing, highlighting their use in test case preparation, bug repair, and debugging. However, the application of LLMs extends beyond general-purpose software testing into specialized domains. Asmita et al. [3] demonstrated how LLMs can enhance fuzzing techniques for embedded systems like BusyBox by generating target-specific initial seeds, significantly increasing crash detection rates. Similarly, Wang et al. [1] presented a system for automating the entire software test process for in-vehicle APIs using LLMs, showcasing their potential in complex industrial settings. These advancements indicate that LLMs are not only automating tedious tasks but also enabling new testing approaches that were previously infeasible due to complexity or resource constraints.

Moreover, LLMs are being leveraged for specific testing challenges such as test oracle generation and bug detection. HAYET et al. [4] introduced ChatAssert, a framework that uses LLMs to generate accurate test oracles by iteratively refining prompts with dynamic and static information. Meanwhile, Li et al. [5] integrated LLMs into static analysis tools to enhance bug detection capabilities, demonstrating their potential to identify practical bugs more effectively. These developments underscore the transformative impact of LLMs on software testing methodologies, offering solutions that improve efficiency, coverage, and reliability across diverse testing scenarios.

II. Recent Applications of RL Testing Methods

Continuous Integration (CI). Reinforcement Learning (RL) addresses key challenges in CI testing by automating and optimizing test case prioritization. Yaraghi et al. [6] developed an RL-based framework that learns from historical execution data, build logs, and code changes to dynamically rank test cases. Their approach outperformed heuristic baselines in early fault detection and regression test efficiency, making it highly suitable for fast-paced CI environments. Bagherzadeh et al. [7] expanded on this by benchmarking PPO, A2C, and DQN across RL frameworks like Stable-baselines and Tensorforce. Their results showed RL's consistent ability to learn optimal execution orders, reducing regression testing costs and adapting to evolving CI pipelines.

Spieker et al. [6] further validated RL's scalability by applying it to large-scale CI systems. Their work demonstrated that RL not only improves prioritization accuracy but also dynamically adjusts test execution in response to real-time code changes. This adaptive approach resulted in higher fault detection rates and more efficient resource use, reinforcing RL's practicality for industrial-grade CI testing workflows.

Autonomous Driving Systems (ADS). Alongside CI, RL has proven instrumental in advancing testing methodologies for autonomous vehicles, particularly through its ability to explore complex, dynamic driving scenarios that are difficult to uncover using heuristic or static approaches. Giamattei et al. [8] demonstrated this by leveraging RL within CARLA simulation environments to adaptively manipulate parameters such as pedestrian behavior, traffic interactions, and environmental conditions. Their method uncovered subtle, safety-critical faults that would likely go undetected through traditional scenario generation. Ma et al. [9] extended this approach through EpiTESTER, integrating RL with epigenetic algorithms and attention mechanisms to guide scenario selection toward high-risk configurations. Their framework significantly improved the discovery of nuanced failure modes in safety-critical ADS software, highlighting RL's capability for intelligent, focused exploration of complex behavior spaces.

Boundary-case validation is another area where RL offers a strategic advantage. Sharifipour et al. [10] applied RL to systematically identify edge-case scenarios by driving agents to the operational limits of ADS systems. Their approach revealed rare transition states associated with failure-prone behaviors, emphasizing RL's strength in stress testing and safety assurance. These works collectively establish RL as a vital tool for high-assurance ADS testing, especially where traditional techniques fall short in depth and adaptability.

Graphical User Interfaces (GUI). Enhancing software testing in highly interactive environments such as GUIs,

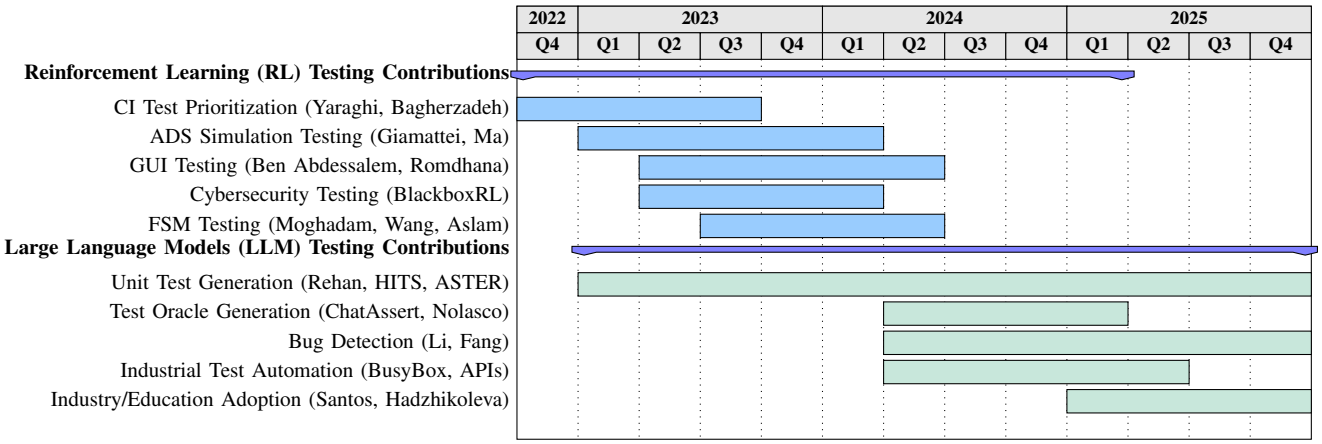

Fig. 1. Timeline of RL and LLM Contributions to Software Testing

RL's exploration accounts for complex, unpredictable user interactions. Yu et al. [11] demonstrated how RL agents can autonomously learn and refine GUI interaction sequences, intelligently selecting actions to maximize functional coverage and fault detection. Their method outperformed traditional heuristic-based approaches by adapting to interface logic during runtime. Romdhana et al. [12] extended these capabilities to vision-based interfaces, training RL agents to interact with GUIs through visual inputs. Their framework enabled the emulation of human-like behavior in dynamic interface layouts, revealing subtle UI faults often missed by fixed-script or pre-recorded strategies. These findings highlight RL's potential to bring perceptual intelligence into GUI testing, allowing for more resilient validation in systems with visual feedback dependencies. Ben Abdessalem et al. [6] reinforced these results by proposing an RL-based framework that adapts its exploration based on GUI state transitions. Their method improved test coverage and discovered faults tied to complex control flow and visual structure. Collectively, these GUI-testing efforts illustrate how RL transcends scripted automation by enabling real-time adaptation to UI logic, making it a valuable tool for robust, coverage-driven interface testing.

Finite State Machines (FSM). FSM-based systems also benefit significantly from RL's ability to manage the complexity of discrete state transitions. Turker et al. [13] applied RL to optimize FSM test-case generation by training agents to prioritize high-reward transition sequences—those likely to lead to untested or fault-prone states. This approach reduced the overhead of exhaustive traversal while substantially improving state coverage and fault detection. By learning which transitions provide the most diagnostic value, RL agents acted as guided heuristics, streamlining the FSM testing process and demonstrating a capacity to internalize software structure. Similarly, Wang et al. [14] employed an adaptive, feedback-driven RL strategy to generate test sequences tailored to uncover state-dependent faults. Their approach yielded notable improvements in fault detection efficiency and software reliability when compared to static or manual FSM-testing techniques.

Altogether, these studies illustrate a clear and compelling case for RL as a domain-agnostic yet deeply effective tool for advancing software testing. Across continuous integration pipelines, autonomous vehicles, mobile and web applications, gaming systems, FSMs, GUI environments, and cybersecurity domains, RL consistently adapts to complex behaviors, learns from structured feedback, and automates exploratory behavior in ways that significantly enhance fault detection, coverage, and scalability. The body of work reviewed here not only confirms RL's unique contributions to improving test generation, prioritization, and execution across diverse software systems, but also establishes a strong empirical foundation for RL's continued evolution as a core component of intelligent software quality assurance.

III. Contributions of Reinforcement Learning Testing Methods

Reinforcement Learning (RL) offers a new model for software testing by transforming test execution from a manually curated process into a goal-oriented, continuously learning system. Unlike heuristic or static rule-based approaches, RL integrates feedback from the system under test to guide exploration, prioritize effort, and refine strategies in real time. This adaptability is especially valuable in complex or rapidly changing environments, where test effectiveness depends not on fixed logic but on the ability to respond to emerging behaviors.

One of RL's most important contributions is its alignment with test objectives. By shaping reward signals around practical goals such as code coverage, early fault detection, or exploration of risky behaviors, RL agents can prioritize actions that increase the value of each test execution. This supports better use of testing resources in environments like continuous integration pipelines or time-constrained development cycles. Instead of following predetermined paths, agents learn to focus attention where it is most needed, which improves fault localization and reduces overhead across builds.

RL also provides significant value in domains with large or unpredictable state spaces, where traditional testing tools fail to generalize. In environments driven by graphical interfaces, FSM logic, or perception-based interaction, RL agents discover transition patterns and interaction sequences that reveal latent defects. This is not the result of brute-force coverage but emerges from learned behaviors that adapt through interaction. RL achieves this without requiring internal models or specifications, which makes it well suited for validating systems that must respond to human input, environmental context, or dynamic configurations.

Another key contribution of RL lies in its resilience to change. Script-based test automation often breaks in response to UI or logic updates. In contrast, RL frameworks adjust dynamically to new states or modified behaviors. This quality reduces the maintenance burden associated with testing infrastructure and supports long-term validation, particularly in agile or iterative development settings. Because agents learn from behavior rather than predefined rules, their strategies remain viable as software evolves.

Finally, RL enhances not just how software is tested, but what is tested. By tying exploration to execution feedback, RL captures behavior patterns that are relevant to users, system stability, or security. In domains where risks are context-sensitive, such as autonomous systems, mobile platforms, or interactive software, RL uncovers failure modes that are difficult to identify manually. In doing so, it shifts the role of testing from structural validation to behavioral assurance, aligning verification with how software is used in real-world conditions.

IV. The Role of Large Language Models in Software Testing

The limitations of traditional testing techniques highlight the urgent need for innovative solutions to streamline and enhance the testing process. Large Language Models (LLMs) have emerged as a transformative technology in software testing. Models such as GPT-4, Llama, and Codex leverage their ability to understand code semantics and generate human-like text to automate and optimize various testing tasks. Research, such as a comprehensive survey of 102 studies on LLMs in software testing [1], underscores their potential to address longstanding challenges. LLMs offer several key advantages:

By generating test cases automatically, LLMs reduce the need for manual effort, accelerating testing cycles. Their contextual understanding enables the creation of diverse test cases, including edge cases that traditional methods often overlook. Automated testing with LLMs provides rapid defect identification, supporting agile development practices. Reducing manual testing lowers overall development costs, making testing more accessible. LLMs can handle the complexity of large-scale systems, adapting to evolving software requirements. This literature review synthesizes insights from 21 research papers published between 2023 and 2025, exploring how LLMs are revolutionizing software testing. It focuses on their applications in unit test case generation, integration testing, system testing, acceptance testing, performance testing, regression testing, security testing, usability testing, fuzz testing, and mutation testing, providing a comprehensive overview of their contributions and the challenges they address.

V. Types of Software Testing in the Software Lifecycle

Understanding the diverse types of software testing conducted throughout the software development lifecycle is essential to appreciating the impact of LLMs. Software testing encompasses a range of activities, each tailored to specific stages of development and deployment, ensuring the software's quality, reliability, and user satisfaction.

1) Unit Testing: The most granular level of testing, unit testing verifies the functionality of individual components or units, such as functions or methods, in isolation. It establishes a foundation for subsequent testing phases by ensuring each unit operates correctly. LLMs can automate unit test generation, producing diverse and high-coverage test cases with minimal manual effort.

2) Integration Testing: This phase tests the interactions between integrated units to identify issues arising from their collaboration. Integration testing is critical for detecting defects in interfaces and data flows. LLMs can assist by generating test scenarios that explore complex interactions and edge cases, enhancing the robustness of integration tests.

3) System Testing: System testing evaluates the complete, integrated system to ensure it meets specified requirements. It includes functional testing to verify features and non-functional testing to assess performance, security, and usability. LLMs can generate comprehensive test suites that cover end-to-end functionality, improving system-level validation.

4) Acceptance Testing: Conducted by end-users or clients, acceptance testing validates that the software fulfills user needs and expectations. Often involving user acceptance testing (UAT), this phase ensures the software is ready for deployment. LLMs can support by generating realistic user scenarios and validating software behavior from a user perspective.

Beyond these core phases, specialized testing types address specific quality aspects:

5) Regression Testing: Ensures that new changes, such as bug fixes or feature additions, do not introduce defects into existing functionality. LLMs can automate the creation and execution of regression test suites, maintaining software stability.

6) Performance Testing: Assesses how the software performs under various conditions, such as high user loads or stress scenarios, to ensure it meets performance expectations. LLMs can simulate load conditions and analyze performance metrics.

7) Security Testing: Identifies vulnerabilities and ensures the software is protected against threats. LLMs can generate test cases targeting potential security weaknesses, enhancing software resilience.

8) Usability Testing: Evaluates the software's user-friendliness and overall user experience. LLMs can analyze user interactions and suggest improvements to enhance usability.

9) Fuzz Testing: Fuzz testing, a specialized form of security testing, involves providing invalid, unexpected, or random inputs to uncover vulnerabilities, such as crashes or memory leaks, that could be exploited. It focuses on stress-testing software to identify edge-case defects, enhancing reliability and security. LLMs can generate diverse, context-aware inputs to trigger vulnerabilities, improving crash detection. This is a key focus of the section on LLMs in Fuzz Testing, where tools like Fuzz4All and studies on Fuzzing BusyBox [3] demonstrate enhanced efficiency across programming languages and embedded systems.

10) Mutation Testing: Mutation testing introduces small changes (mutants) into the code to evaluate the effectiveness of test suites, ensuring they can detect these changes and thus assess test suite quality. It is crucial for verifying the robustness of tests, particularly in unit and regression testing. LLMs can generate effective test cases targeting mutants, improving mutation scores and detecting subtle bugs. LLMs can be integrated across these testing types to improve efficiency, coverage, and accuracy. For example, in unit testing, tools like ASTER [15] leverage LLMs to generate natural and high-coverage tests for Java and Python. In regression testing, frameworks like CoverUP [16] achieve up to 90% coverage by iteratively refining tests. By applying LLMs to these phases, development teams can achieve more robust and efficient testing practices, as detailed in the subsequent sections of this review.

VI. Challenges Addressed by LLMs in Software Testing

LLMs present a promising remedy by leveraging their deep understanding of both code syntax and semantics. Their generative capabilities allow them to propose diverse test scenarios, including those that would be impractical or overlooked through manual specification. By learning from historical patterns and user intent, LLMs can proactively suggest high-risk or previously untested code paths, potentially achieving near-complete coverage with minimal human intervention.

Another critical pain point is the time-consuming nature of manual testing, which slows iteration cycles and burdens developers. LLMs can significantly accelerate this process by automatically generating test suites, documentation, and even commentary for test behavior. This not only reduces the overhead associated with test creation but also enables continuous testing as code evolves. Additionally, the high costs associated with building and maintaining large QA teams can be alleviated through LLM automation. By reducing the need for manual labor and optimizing test selection and reuse, organizations can drive down operational expenses while improving test consistency.

Scalability also emerges as a pressing concern in large systems, where traditional testing techniques become infeasible due to the sheer volume of interactions and modules. LLMs are naturally equipped to address this challenge by processing large-scale codebases and adapting across diverse programming paradigms. Their ability to generalize allows them to generate and maintain tests for complex enterprise software, distributed systems, or rapidly evolving microservices architectures.

Finally, the test oracle problem determining the correct expected output remains a conceptual barrier for many testing frameworks. Here, LLMs provide a novel advantage: their contextual awareness and pattern recognition can infer likely outputs based on learned behavior, documentation, and user-defined specifications. As they continue to evolve through feedback loops, LLMs may serve as intelligent oracles capable of not just validating test outcomes but also suggesting corrections and refinements to both the test and the code under test.

In sum, LLMs offer a transformative shift in how software testing challenges are approached. By aligning their strengths scalability, generation, semantic understanding, and adaptiveness with the limitations of existing methods, they present a unified and automated pathway toward more robust, efficient, and intelligent software quality assurance.

TABLE I
Challenges in Software Testing and Potential of LLMs

Challenge	Description	Potential of LLMs
Low Test Coverage	Traditional methods miss edge cases and complex scenarios.	LLMs could generate test cases that cover a broader range of scenarios, including rare edge cases, by leveraging their understanding of code and natural language. They might also learn from historical data to identify and prioritize previously uncovered cases, ensuring near-total coverage without human intervention.
Time-Consuming Processes	Manual testing slows development cycles.	LLMs could generate test cases that cover a broader range of scenarios, including rare edge cases, by leveraging their understanding of code and natural language. They might also learn from historical data to identify and prioritize previously uncovered cases, ensuring near-total coverage without human intervention.
High Costs	Testing needs significant resources.	By fully automating testing tasks, LLMs could drastically reduce the need for large testing teams, lowering labor costs. Additionally, LLMs might optimize resource usage by generating efficient test cases that require fewer computational resources, making testing more cost-effective for small and large organizations alike.
Scalability Issues	Traditional methods fail with large systems.	LLMs could handle even larger and more complex systems by processing vast amounts of code and generating tests at scale. Their adaptability to different programming languages, frameworks, and domains could make them a universal solution for diverse software projects, from embedded systems to enterprise applications.
Test Oracle Problem	Determining correct outputs is hard.	LLMs could potentially generate highly accurate test oracles by deeply understanding code intent, expected behavior, and domain-specific requirements. They might also learn from past test results and adapt over time, improving oracle quality and reducing the need for manual validation.

VII. Contributions of LLMs in Software Testing

The reviewed papers showcase significant contributions of LLMs across various testing domains:

A. LLMs in Unit Testing

Large Language Models (LLMs) are transforming unit testing by automating test case generation while addressing the inefficiencies of manual testing such as its time-consuming nature and limited coverage. By leveraging their ability to comprehend code, LLMs generate diverse, high-coverage tests which enhance both scalability and efficiency. This automation not only reduces manual effort but also enables thorough testing of large-scale systems, as demonstrated by industrial applications like Meta's ACH system [17].

Research underscores the effectiveness of LLMs in unit testing across various contexts. For instance, Rehan et al. (2025) used the Llama-2 model with QLoRA quantization to generate unit tests for Java methods, achieving high coverage while reducing redundancy through human-in-the-loop validation [2]. A survey by Wang et al. (2024) of 102 studies revealed that LLMs like Codex and ChatGPT are widely used for test generation, with coverage rates ranging from 2% to 89% by using prompt engineering and fine-tuning strategies [1]. Tools like HITS [18], ASTER [15], and CoverUP [16] further illustrate LLMs' potential to achieve superior coverage through innovative techniques such as method decomposition, static analysis integration and coverage-guided prompting. In industry, Meta's ACH system leverages the Llama 3.1 70Bn model for mutation-guided test generation with 73% of tests accepted by engineers and 36% deemed privacy-relevant [17]. Additionally, tools like MuTAP enhance bug detection by augmenting LLM prompts with surviving mutants [19]. Studies by Fang et al. (2025) and Hossain et al. (2024) reinforce LLMs' strengths in code analysis and automated program repair, respectively [20], [21].

Despite these advancements, challenges remain. As noted by Rehan et al. (2025), generated tests often require human validation for correctness [2]. LLMs may also produce tests with syntactic errors which in turn necessitates repair mechanisms, as seen in ASTER's approach [15]. Ensuring tests target critical code paths is another hurdle, though CoverUP's coverage-guided prompting offers a partial solution [16]. Furthermore, the computational demands of fine-tuning LLMs, as highlighted by MuTAP, can limit their applicability in resource-constrained environments [19]. These findings highlight LLMs' transformative potential in unit testing while also pointing to areas needing further refinement.

B. LLMs in Regression Testing

Large Language Models (LLMs) are reshaping regression testing by tackling the challenges like time-intensive processes and limited coverage through automated test case generation and improved fault detection. Regression testing often demands significant resources as it is essential to confirm that software updates or bug fixes do not disrupt existing functionalities. LLMs address this by leveraging their code analysis capabilities to create comprehensive, targeted test cases which reduce manual effort and enable scalable testing for complex systems. This automation marks a significant step forward in enhancing software quality assurance.

Notable tools demonstrate LLMs' impact in this domain. The CoverUP framework, for instance, employs LLMs like OpenAI's GPT-4o to generate high-coverage Python regression tests by using coverage analysis to identify untested code and iteratively refine tests. It achieves a per-module median line+branch coverage of 80% and an overall coverage of 90% which significantly outperforms traditional tools like CoDAMosA (47%) and MuTAP (77%) [16]. Similarly, TOGLL, while focused on test oracle generation, bolsters regression testing by producing oracles that detect 1,023 unique mutants missed by EvoSuite which enhances test suite reliability [22]. Tools like TESTPILOT for JavaScript and ChatUniTest for Java further contribute by generating unit tests that integrate into regression suites. A survey by Santos et al. indicates that 28% of testing professionals use LLMs for test design and execution which reflects growing industry adoption [23]. However, challenges persist, including difficulties integrating LLMs into existing workflows, managing large test suites and ensuring test relevance. The survey advises cautious adoption due to the nascent stage of LLM use in testing and the need for standardized guidelines [23]. Despite these hurdles, LLMs' ability to streamline and enhance regression testing highlights their potential to redefine software testing practices.

C. LLMs in System Testing

System testing plays a vital role in software development. It ensures that an integrated system meets both functional requirements, like correct behavior and non-functional ones such as performance and security. Traditional system testing, however, is often labor-intensive and struggles to cover the full range of scenarios; particularly in large-scale or complex systems. Large Language Models (LLMs) offer a promising solution by automating the generation of diverse test inputs that enhances bug detection and improves overall testing efficiency; thereby addressing these longstanding challenges.

LLMs contribute significantly to system testing through several key capabilities. They excel at generating varied and realistic test inputs, which are essential for thoroughly testing complex systems like mobile applications and deep learning libraries. For instance, the Fuzz4ALL tool employs a dual-LLM architecture to create test inputs across multiple programming languages which effectively identifies edge cases and boosts test coverage. In deep learning library testing, LLMs have achieved up to 66% API coverage for TensorFlow that demonstrates their ability to produce comprehensive test scenarios [1]. Additionally, LLMs automate test execution, particularly in mobile app testing, where iterative prompting refines test cases to reduce manual effort. They also enhance bug detection by uncovering issues in sophisticated systems like SMT solvers and compilers, often through differential testing techniques. Furthermore, tools like LogLLM leverage LLMs for log-based anomaly detection by analyzing system logs with high accuracy to monitor system health and complement traditional testing efforts [24].

Despite these advancements, LLMs face notable challenges in system testing. Some studies report low coverage rates, such

as 2% line coverage in specific benchmarks which indicates that LLMs may not always achieve the desired thoroughness [1]. The oracle problem—difficulty in determining correct outputs for generated inputs—remains a significant hurdle, though focusing on crash bugs offers a partial workaround. Moreover, the high computational resources required to run LLMs and concerns about data privacy can limit their practical adoption in real-world settings. These challenges highlight the need for careful implementation and further research to optimize LLM performance.

D. LLMs in Acceptance Testing

Acceptance testing validates that the software meets user requirements and is ready for deployment which often involves end-users to ensure practical usability. Traditional acceptance testing relies on subjective user feedback, making automation challenging and time consuming. Although the reviewed research papers do not directly address LLMs in acceptance testing, their natural language processing capabilities could potentially generate user scenarios and validate software behavior. It results in improved efficiency and alignment with user needs.

Automation of acceptance testing using LLMs is challenging because acceptance testing relies on domain-specific knowledge and human judgment. Ensuring alignment with user expectations and handling vague requirements are significant hurdles.

E. LLMs in Performance Testing

Performance testing assesses software behavior under various conditions, such as high user loads, to ensure it meets performance expectations like response time and throughput. Traditional performance testing is resource-intensive and requires specialized setups which limits scalability. While, the reviewed research papers do not provide specific approaches for LLMs in performance testing, their potential to simulate workloads and analyze metrics suggests future applications, which could address these challenges.

Performance testing requires specialized methods for load simulation and metric evaluation, which are not addressed by current LLM capabilities. Data scarcity and the need for real-world performance benchmarks pose additional challenges.

F. LLMs in Security Testing

Security testing plays a pivotal role in protecting software systems from vulnerabilities that could lead to severe threats, such as data breaches. Traditional methods like penetration testing are often labor-intensive and struggle to address the full spectrum of potential attack vectors which leaves systems exposed to risks. Large Language Models (LLMs) offer a transformative approach by enhancing security testing through techniques like fuzz testing and advanced bug detection. LLMs can uncover vulnerabilities that conventional methods might overlook by generating diverse and targeted inputs. It improves both the efficiency and effectiveness of security testing processes.

Several innovative tools and frameworks demonstrate the potential of LLMs in this domain. Fuzz4All employs LLMs to generate a broad array of inputs, effectively identifying vulnerabilities across various software systems [25]. TOGLL strengthens bug detection, particularly for security-related issues, by leveraging robust oracles to validate test outcomes. Meta's ACH framework utilizes LLMs to produce tests that fortify code against privacy-related regressions, with 36% of generated tests deemed privacy-relevant [17]. The LLiFT framework excels at detecting Use Before Initialization (UBI) bugs in complex systems like the Linux kernel which achieved 100% recall [5]. In specialized applications, SPAPI-Tester achieves 96% accuracy in identifying buggy APIs [26] in specialized applications while the AID framework detects complex bugs with 85.09% precision. These tools highlight LLMs' versatility in addressing diverse security testing needs.

The contributions of LLMs to security testing are multifaceted. As evidenced by Fuzz4All and LLiFT, LLMs significantly enhance vulnerability detection. LLMs also streamline test generation which is demonstrated by Meta's ACH which produces privacy-hardening tests that address critical concerns. Automation is another key benefit, SPAPI-Tester's exemplifies another key benefit of automation by detecting 22 bugs across 193 APIs that results in significant reduction in manual effort. However, challenges persist. The opaque, black-box nature of LLMs complicates explainability, a critical requirement in security testing where understanding the rationale behind test generation is essential. Precision remains an issue in some cases, with certain AID applications reporting precision as low as 6.3%. Additionally, the complexity and diversity of attack vectors pose ongoing challenges which necessitates further advancements to ensure comprehensive coverage.

G. LLMs in Usability Testing

Usability testing ensuring intuitive navigation and accessibility by evaluating user-friendliness and overall user experience of a software. Automation in this domain is challenging because of the reliance of traditional usability testing on subjective user feedback. The lack of usability-specific datasets also acts as a barrier. The reviewed research papers do not mention specific LLM approaches for usability testing, but their ability to analyze user interactions could potentially support evaluation, addressing efficiency and scalability challenges.

H. LLMs in Fuzz Testing

Fuzz testing, commonly known as fuzzing, is a vital technique for identifying software vulnerabilities by supplying programs with invalid, unexpected or random inputs. Traditional fuzzing methods often fall short in generating structured inputs that align with a program's specific grammar which results in limited code coverage and missed vulnerabilities. Large Language Models (LLMs) are revolutionizing fuzz testing by producing diverse, context-aware inputs that enhance crash detection and improve coverage. LLMs address these shortcomings by leveraging their deep understanding of code and

documentation which makes fuzz testing more effective across various software domains.

The impact of LLMs in fuzz testing is evident through several innovative tools and studies. Fuzz4All, a universal fuzzing framework, employs a dual-LLM architecture: a *distillation LLM* processes user-provided documentation or code examples to create prompts, while a *generation LLM* produces test inputs across multiple programming languages. This approach allows Fuzz4All to trigger edge cases and uncover vulnerabilities more efficiently than traditional fuzzing methods, particularly for applications requiring diverse input formats [25]. In embedded systems, the *Fuzzing BusyBox* study demonstrated LLMs' effectiveness by generating target-specific initial seeds for mutation-based, coverage-guided fuzzing with tools like AFL++. These seeds led to more crashes and greater code coverage compared to random seeds which highlights LLMs' value in resource-constrained environments [3]. The study also introduced a crash reuse strategy, where LLMs reuse crash data to optimize fuzzing which saves time and resources. Other frameworks, such as mGPTFuzz [27] and FuzzGPT [28], further showcase LLMs' capabilities in input generation and mutation for security testing. For instance, Deng et al.'s work combines generative and infilling LLMs with mutation operators to fuzz deep learning libraries which significantly increases input diversity [1]. Similarly, Hu et al.'s greybox fuzzer augments seed mutation to explore new code regions, while Xia et al.'s Fuzz4All incorporates auto-prompting and an LLM-powered fuzzing loop [25]. These examples underscore LLMs' ability to generate high-quality, structured inputs that traditional fuzzing struggles to produce.

Despite their transformative potential, LLMs in fuzz testing face several challenges. Ensuring the quality of generated seeds is critical, as poor seeds can lead to inefficient fuzzing which is highlighted by Stuart's Law (MSL). The computational costs of LLM inference also pose a significant barrier in resource-constrained settings. A survey by Wang et al. notes that achieving high code coverage remains a challenge which suggests that complementary techniques, such as mutation testing, are needed to enhance input diversity [1]. Integrating LLMs into existing fuzzing workflows requires careful calibration to balance their strengths with practical constraints, such as resource demands and the need for high-quality prompts. Overall, LLMs are reshaping fuzz testing by addressing the limitations of traditional methods through context-aware input generation and improved code coverage. As LLMs continue to advance, their role in fuzz testing is poised to grow by offering more robust and efficient methods to enhance software security.

I. LLMs in Mutation Testing

Mutation testing serves as a robust method to gauge a test suite's strength by injecting small, deliberate changes(mutants) into code and checking whether tests detect these alterations. Traditionally, this process demands significant manual effort, from crafting test cases to validating mutants, especially equivalent ones that preserve program behavior. Large Language Models (LLMs) are remodeling this field by automating test case generation, enhancing equivalent mutant detection and enabling scalable testing. This automation reduces manual labor and elevates the reliability of software testing.

Several tools highlight LLMs' transformative impact. TOGLL evaluated with the PIT mutation testing tool across 25 Java projects and detecting 1,023 unique mutants missed by EvoSuite. It outperformed TOGA by nearly tenfold [22]. ChatAssert leverages ChatGPT, with a prompt engineering framework incorporating code summarization and repair tools which produces test oracles that surpass TECO by 15% in accuracy [4]. MuTAP, a mutation-guided approach, augments LLM prompts with surviving mutants and achieves a 93.57% mutation score on synthetic buggy Python code and detecting 28% more faults than Pynguin [19]. In industry, Meta's Automated Compliance Hardener (ACH) uses LLMs (Llama 3.1 70Bn) to generate 571 privacy-hardening tests across 10,795 Android Kotlin classes, with 73% accepted by engineers and 36% deemed privacy-relevant [29]. Molina et al. (2025) review LLM-based oracle generation.However, it notes challenges like data biases and reproducibility while proposing a research roadmap [30]. These advancements underscore LLMs' ability to enhance test suite effectiveness across diverse contexts. Detecting equivalent mutants is complex as it often requires advanced validation to distinguish them from undetected flaws. The computational demands of LLMs, particularly for training and inference, can strain resources which limits applicability in some settings. Ensuring test case diversity to address varied mutants is another hurdle. A survey by Santos et al. (2025) advocates cautious adoption and emphasizes the need for robust guidelines to maximize LLMs' benefits [23].

VIII. RL vs. LLMs: Comparative Analysis of Utility, Architectural Correlations, and Contextual Methodology Selection

Reinforcement Learning (RL) and Large Language Models (LLMs) have both revolutionized software testing methodologies, automating tasks traditionally dominated by manual or heuristic approaches. Despite their distinct underlying mechanisms, both methodologies have demonstrated substantial improvements in efficiency, adaptability, and accuracy across various testing domains. This section provides a comparative analysis of RL and LLM methodologies, highlights the correlations between their architectural traits and domain-specific successes, and proposes contexts in which each approach is optimally suited.

A. Automation of Test Generation

Automated test generation has become critical for improving testing efficiency and reducing manual oversight in software engineering. Both RL and LLM approaches offer innovative solutions to automating this crucial aspect. RL can dynamically generate and prioritize test cases by learning optimal testing strategies from real-time interactions and historical execution data. This approach significantly mitigates human bias and ensures comprehensive test coverage, particularly

in continuous integration environments (Yaraghi et al. [6], Bagherzadeh et al. [7]). In many cases, these automatically generated test cases outperformed those that were employed with heuristic and manual testing strategies. In situations where there was access to historical data, RL methods tended to perform incredibly well compared to heuristic and manual tests. Importantly, RL tended to perform well in highly dynamic environments, which is valuable since real-life software is constantly being changed daily, with new tests needing to be written frequently. However, these RL-generated test policies can be difficult for humans to interpret, lowering their readability, maintainability, and potentially complicating their integration into established workflows. This limitation makes RL less suitable for situations where clarity and ease of integration is paramount, such as in codebases. In contrast, LLMs automate test case generation by leveraging extensive syntactic and semantic knowledge of code repositories and static analysis techniques (Rehan et al. [2]; Wang et al., [1]; Pan et al. [15]). These models produce highly readable and maintainable tests, facilitating easier human validation and integration into traditional software engineering practices. Nevertheless, LLMs lack adaptive real-time feedback mechanisms, making them suboptimal for dynamically changing or highly interactive environments. In summary, both RL and LLM enhance test generation automation but are optimally suited for distinctly different operational contexts, balancing dynamic adaptability with static analytical rigor. RL being advantageous in environments demanding interactive, real-time adaptability whereas LLMs excelling in predictable, structured scenarios requiring detailed semantic and syntactic understanding.

B. Utility in Specialized and Domain-Specific Applications

The complexity and specificity of certain domains require highly tailored methodologies with deep contextual demands. RL and LLM have demonstrated significant utility across various specialized fields. RL has proven highly effective in specialized domains such as autonomous driving systems (Giamattei et al. [8], Ma et al. [9]), leveraging simulation environments to create adaptive, safety-critical test scenarios. RL excels in these situations as it benefits from its exploratory nature, which LLMs typically lack. For example, Giamattei et al. [8] showcased RL's ability to identify subtle, but critical faults, exemplifying its comprehensive testing abilities. RL techniques prove particularly beneficial in specialized domains, where they generate adaptive, safety-critical test scenarios through simulation interactions and continuous learning. However, RL's compute heavy demands and long training periods can be barriers to efficient utilization, whereas LLMs can be applied relatively quickly with a pre-trained model. Conversely, LLMs excel in specialized contexts involving structured, mostly static software environments like embedded systems and automotive API testing contexts, utilizing their extensive domain knowledge and generative abilities to deliver highly targeted test artifacts (Wang et al. [14], Asmita et al. [3]). Their generative capabilities effectively utilize extensive domain-specific data to produce precise and targeted test cases. Yet, LLMs may struggle when encountering environments with limited or insufficient historical data, resulting in decreased efficacy. RL's interactive adaptability excels in dynamic environments that demand interactive, real-time adaptability and exploration like autonomous systems. Whereas LLM's strength lies in structured domains with predefined contexts where deep semantic understanding significantly enhances testing precision. Thus, each methodology uniquely addresses the needs of specialized domains through either dynamic adaptability or detailed domain knowledge integration.

C. Bug Detection, Security Applications, and Static Analysis Integration

Effectively identifying software defects and vulnerabilities is paramount to software reliability and security. Advanced methodologies like RL and LLM provide innovative approaches to bug detection and cybersecurity testing. RL methods like BlackboxRL actively adapt to system responses, uncovering vulnerabilities through adaptive penetration testing and dynamic interaction, substantially improving over static methods (Romdhana et al. [12]). This is particularly notable as cybersecurity testing typically utilizes static and random methods of testing, failing to explore enough to find some harder to find vulnerabilities. These techniques leverage dynamic feedback from the software environment, continuously adjusting their testing strategy to uncover deep, subtle vulnerabilities that static or heuristic methods typically miss. LLM methodologies significantly enhance static analysis for bug detection and vulnerability assessment. Methods proposed by Li et al. [5] and Fang et al. [20] integrate semantic interpretation capabilities of LLMs with traditional static analyzers, enabling deeper identification of practical defects and subtle security vulnerabilities. This hybrid approach provides more nuanced insights than purely static methods alone. RL's feedback-driven search reveals dynamic, emergent vulnerabilities, but is compute intensive and results can be opaque. LLM methodologies are fast and interpretable with static analysis frameworks, effectively handling complex, semantic-dependent defect detection. However, LLMs are blind to runtime phenomena that are discovered via direct interaction, so they may fail to uncover the same level of depth that RL is capable of achieving. Ultimately, RL and LLM methodologies complement each other, covering dynamic penetration testing and semantic static analysis comprehensively.

D. Testing Coverage in Complex Systems

Ensuring comprehensive test coverage is particularly challenging in complex systems characterized by vast state spaces and intricate interactions. Both RL and LLM methodologies significantly improve coverage capabilities. RL methodologies have been shown to effectively navigate and manage complex interaction scenarios, adaptively exploring and identifying critical state transitions to detect otherwise elusive faults. For example, Yu et al. [11] introduced an RL-driven approach that incrementally learns effective GUI interaction

sequences , leading to significantly improved coverage in GUI systems. By systematically covering complex states and transitions through adaptive strategies, RL substantially enhances fault detection capabilities. LLM techniques employ structured static analysis and method slicing to improve testing coverage systematically. For instance, Wang et al. [18] (HITS) focus on decomposing intricate Java methods into distinct, smaller slices, each independently analyzed and tested to maximize branch coverage. Pan et al. [15] (ASTER) adopts a language-agnostic pipeline, where models parse and reformulate both Java and Python code segments into discrete units, enabling comprehensive exploration of high-risk functional paths. By iteratively refining these smaller components, LLMs can accurately assess dependencies, capture edge cases, and guarantee coverage of critical pathways. This structured analytical approach significantly enhances coverage accuracy and overall test comprehensiveness, even in complex software systems. RL is particularly suited for adaptive exploration in dynamic, interaction-rich environments. LLM approaches excel in structured, predictable contexts by systematically decomposing and analyzing complex software components to ensure comprehensive testing coverage. RL dominates in cases where interaction order matters, LLMs excel when structural reasoning suffices. Both RL and LLM methodologies significantly enhance testing coverage in complex systems, each offering distinct advantages that collectively improve overall software testing effectiveness.

IX. Transformational Impact on the Software Testing Paradigm

The introduction of RL and LLM methodologies represents a significant transformation in software testing, fundamentally altering traditional testing paradigms by introducing advanced adaptive and generative capabilities. RL methodologies represent a transformative shift away from static, predefined test methods towards dynamic, adaptive strategies driven by continuous learning and environmental feedback. For example, Yaraghi et al. [6] utilized RL in continuous integration pipelines, observing that adaptive test prioritization swiftly identified breaking changes. Bagherzadeh et al. [7] compared the efficacy of diverse RL frameworks, showcasing how learned policies reduce testing overhead by focusing on historically failure-prone areas. By embracing exploration and iterative refinement, RL effectively reshapes how tests are scheduled, executed, and analyzed in evolving software ecosystems. LLM methodologies revolutionize software testing by automating extensive aspects of the test generation lifecycle, such as creating unit tests, test oracles, and bug detection reports (Santos et al. [23], Wang et al. [14]). Santos et al.'s [23] industry survey noted that LLMs frequently assist in test design and debugging, indicating widespread acceptance of these generative approaches. Meanwhile, Wang et al. [14] illustrated how LLM-based modules orchestrate comprehensive testing tasks in automotive software, exemplifying the profound shift towards automated code analysis and test artifact generation. RL methodologies fundamentally transform testing through adaptive, interactive learning capabilities suitable for dynamic environments, whereas LLM methodologies provide revolutionary improvements through automated generative capabilities ideal for structured, predictable testing scenarios. In conclusion, RL and LLM methodologies each offer transformative contributions to software testing, providing powerful complementary approaches that comprehensively address current and emerging software testing challenges.

X. Choosing which method to use

When deciding whether to use either RL or LLMs for either research or practical industry applications, there are many key components to consider to determine which is the best fit. Selecting between RL and LLMs involves careful evaluation of various factors, including environmental dynamics, available data, complexity, computational resources, interpretability, exploratory capabilities, and time constraints.

Since the decision hinges on the nature of the task, the type of environment, and the goals of the system, it has been noted that RL excels in dynamic and interactive environments where agents must adapt continuously based on feedback. It thrives in applications that require real-time decision-making under uncertainty, such as robotics, autonomous systems, or adaptive software testing, where constant exploration and response to evolving conditions are essential. In contrast, LLMs are more suited to well-defined, data-rich domains, particularly when working with static historical data, such as code repositories, log files, or structured documentation. They demonstrate remarkable fluency in domains where semantic understanding, contextual reasoning, or knowledge generation is critical.

From a resource perspective, RL demands extensive simulation environments and training time, often accompanied by high computational costs due to repeated learning episodes. LLMs, although computationally heavy during training and inference, may offer a more plug-and-play solution if a sufficiently large pre-trained model is available, reducing startup costs in projects where inference speed is acceptable. Furthermore, when it comes to interpretability, RL strategies are often opaque, making it difficult to trace or explain specific decisions, especially in stochastic environments. LLMs, while also not entirely interpretable, produce readable and maintainable outputs, which can enhance debugging and system transparency.

In terms of exploration and adaptability, RL agents shine in tasks where agents must navigate partially observable or novel states, progressively refining their strategies. LLMs, by contrast, are optimized for contextual generalization and semantic synthesis—making them especially powerful when depth of knowledge and domain fluency are more valuable than continuous adaptation. Finally, time constraints also influence the selection: RL's iterative nature may delay deployment due to long training cycles, whereas LLM-based methods can achieve rapid prototyping and fine-tuning, particularly when building on existing foundation models.

Given the cost and benefit found in each domain of software testing methods, practitioners should evaluate their project-

TABLE II
SAMPLE 3-COLUMN TABLE (SINGLE COLUMN FIT)

Factor	RL	LLM
Environment	Dynamic/Interactive with real-time feedback	Well-defined domain with extensive static data
Data Availability/Type	Feedback loops are present	Large amounts of textual/domain-specific data
Complexity	Handles complicated, changing states well	Complex code or Large Semantic Knowledge
Computational Resources	Heavy compute available for simulations	Requires large amounts of memory, but can reuse pre-trained models
Interpretability	Policies can be difficult to understand	Generated results are highly readable
Exploratory Capability	Excels at exploring and finding new strategies/states	Great for applying deep knowledge with static semantic context
Resource Trade-Offs	Training can be long	Fine-tuning is quick if a large model is already available

specific constraints against these considerations to effectively harness the strengths of each methodology to maximize overall testing effectiveness and efficiency.

XI. CONCLUSION

Large Language Models (LLMs) and Reinforcement Learning (RL) are revolutionizing software testing by automating and enhancing quality assurance. LLMs excel in generating high-coverage test cases and accurate oracles, particularly in code-centric domains. Their versatility shines in specialized contexts like embedded systems and automotive APIs, though challenges like functional inaccuracies and integration complexities persist. Conversely, RL thrives in dynamic environments, prioritizing tests and uncovering subtle faults in continuous integration, autonomous driving, and cybersecurity. RL's exploratory nature navigates complex state spaces, yet high computational demands and interpretability issues remain barriers. LLMs suit predictable, code-based tasks, while RL excels in interactive, state-rich scenarios, suggesting hybrid approaches where LLMs generate tests and RL refines them could enhance efficiency. Future research should focus on standardizing integration, improving RL interpretability, and developing resource-efficient methods. By addressing these limitations, LLMs and RL can become foundational to intelligent software testing, ensuring reliability and security in modern systems through automation and adaptability.

REFERENCES

[1] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Trans. Softw. Eng.*, vol. 50, no. 4, p. 911–936, Apr. 2024. [Online]. Available: https://doi.org/10.1109/TSE.2024.3368208

[2] S. Rehan, B. Al-Bander, and A. Al-Said Ahmad, "Harnessing large language models for automated software testing: A leap towards scalable test case generation," *Electronics*, vol. 14, no. 7, 2025. [Online]. Available: https://www.mdpi.com/2079-9292/14/7/1463

[3] Asmita, Y. Oliinyk, M. Scott, R. Tsang, C. Fang, and H. Homayoun, "Fuzzing BusyBox: Leveraging LLM and crash reuse for embedded bug unearthing," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 883–900. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/asmita

[4] I. Hayet, A. Scott, and M. dx27;Amorim, " ChatAssert: LLM-Based Test Oracle Generation With External Tools Assistance ," *IEEE Transactions on Software Engineering*, vol. 51, no. 01, pp. 305–319, Jan. 2025. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TSE.2024.3519159

[5] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3649828

[6] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. C. Briand, "Scalable and accurate test case prioritization in continuous integration contexts," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1615–1639, 2023.

[7] B. M. G. T. e. a. Pan, R., "Test case selection and prioritization using machine learning: a systematic literature review - Empirical Software Engineering — link.springer.com," 2021, [Accessed 23-04-2025].

[8] B. M. P. R. e. a. Giamattei, L., "Reinforcement learning for online testing of autonomous driving systems: a replication and extension study," 2025, [Accessed 23-03-2025].

[9] A. S. . Y. T. Ma, T., "Testing self-healing cyber-physical systems under uncertainty with reinforcement learning: an empirical study - Empirical Software Engineering — link.springer.com," 2021, [Accessed 23-03-2025].

[10] M. Biagiola and P. Tonella, "Boundary state generation for testing and improvement of autonomous driving systems," *IEEE Transactions on Software Engineering*, vol. 50, no. 8, pp. 2040–2053, 2024.

[11] S. Yu, C. Fang, X. Li, Y. Ling, Z. Chen, and Z. Su, "Effective, platform-independent gui testing via image embedding and reinforcement learning," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 7, Sep. 2024. [Online]. Available: https://doi.org/10.1145/3674728

[12] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep reinforcement learning for black-box testing of android apps," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, Jul. 2022. [Online]. Available: https://doi.org/10.1145/3502868

[13] U. C. Türker, R. M. Hierons, K. El-Fakih, M. R. Mousavi, and I. Y. Tyukin, "Accelerating finite state machine-based testing using reinforcement learning," *IEEE Transactions on Software Engineering*, vol. 50, no. 3, pp. 574–597, 2024.

[14] S. Wang, Y. Yu, R. Feldt, and D. Parthasarathy, "Automating a complete software test process using llms: An automotive case study," 2025. [Online]. Available: https://arxiv.org/abs/2502.04008

[15] R. Pan, M. Kim, R. Krishna, R. Pavuluri, and S. Sinha, "Aster: Natural and multi-language unit test generation with llms," 2025. [Online]. Available: https://arxiv.org/abs/2409.03093

[16] J. A. Pizzorno and E. D. Berger, "Coverup: Coverage-guided llm-based test generation," 2025. [Online]. Available: https://arxiv.org/abs/2403.16218

[17] C. Foster, A. Gulati, M. Harman, I. Harper, K. Mao, J. Ritchey, H. Robert, and S. Sengupta, "Mutation-guided llm-based test generation at meta," 2025. [Online]. Available: https://arxiv.org/abs/2501.12862

[18] Z. Wang, K. Liu, G. Li, and Z. Jin, "Hits: High-coverage llm-based unit test generation via method slicing," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1258–1268. [Online]. Available: https://doi.org/10.1145/3691620.3695501

[19] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, "Effective test generation using pre-trained large language models and mutation testing," *Information and Software Technology*, vol. 171, p. 107468, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584924000739

[20] C. Fang, N. Miao, S. Srivastav, J. Liu, R. Zhang, R. Fang, Asmita, R. Tsang, N. Nazari, H. Wang, and H. Homayoun, "Large language models for code analysis: do llms really do their job?" in *Proceedings of the 33rd USENIX Conference on Security Symposium*, ser. SEC '24. USA: USENIX Association, 2024.

[21] S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W.-H. Chiang, Y. Lyu, H. Nguyen, and O. Tripp, "A deep dive into large language models for

automated bug localization and repair," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660773

[22] S. B. Hossain and M. Dwyer, " TOGLL: Correct and Strong Test Oracle Generation with LLMs ," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 635–635. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00098

[23] R. Santos, I. Santos, C. Magalhaes, and R. de Souza Santos, "Are we testing or being tested? exploring the practical applications of large language models in software testing," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2024, pp. 353–360.

[24] W. Guan, J. Cao, S. Qian, J. Gao, and C. Ouyang, "Logllm: Log-based anomaly detection using large language models," 2025. [Online]. Available: https://arxiv.org/abs/2411.08561

[25] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639121

[26] "Amazon Selling Partner API (SP-API) — developer.amazonservices.com," https://developer.amazonservices.com/, [Accessed 25-04-2025].

[27] X. Ma, L. Luo, and Q. Zeng, "From one thousand pages of specification to unveiling hidden bugs: Large language model assisted fuzzing of matter IoT devices," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 4783–4800. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/ma-xiaoyue

[28] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623343

[29] C. Foster, A. Gulati, M. Harman, I. Harper, K. Mao, J. Ritchey, H. Robert, and S. Sengupta, "Mutation-guided llm-based test generation at meta," 2025. [Online]. Available: https://arxiv.org/abs/2501.12862

[30] F. Molina, A. Gorla, and M. d'Amorim, "Test oracle automation in the era of llms," *ACM Trans. Softw. Eng. Methodol.*, Jan. 2025, just Accepted. [Online]. Available: https://doi.org/10.1145/3715107