International Journal of Scientific Research & Engineering Trends
Volume 12, Issue 6, Nov-Dec-2026, ISSN (Online): 2395-566X

LLM-Based Unit Test Generation: A Survey of Approaches, Evaluation Practices, And Reporting Gaps

Khushal Soni
M.Tech Software Engineering, Semester I, Gandhinagar Institute Of Technology, Gujarat, India
Khushalsoni134@Gmail.Com | November 2026

Abstract- Unit tests are cheap to run but expensive to write, and large language models (LLMs) have become a popular way to write them automatically. Papers on the topic now use different languages, benchmarks and metrics, so their headline numbers cannot be lined up against each other. This survey reviews nine LLM-based unit test generation approaches published between 2023 and 2026, together with one benchmark and three earlier surveys. It groups the approaches into five families: prompt-only generation, generate-validate-repair loops, analysis or coverage-guided generation, hybrids with search-based testing, and industrial pipelines. For each study it records the setting, the technique and the result as reported by the authors, and it separates what the authors measured from what this survey interprets. The comparison shows that coverage is the most common metric, that mutation score is rarely central, that human or developer judgement appears in only a few studies, and that closed commercial models make some results hard to repeat. A small but telling data point is that, in one study, only 24.8% of the tests written by default ChatGPT passed when executed. The paper closes with a reporting checklist and a list of research directions. It is a literature survey, and no experiments were run.

Keywords: unit test generation, large language models, software testing, code coverage, mutation score, benchmarks, survey.

I. INTRODUCTION

Unit testing checks small pieces of code, usually single functions or classes, and catches regressions early. Writing good unit tests takes a lot of developer time. Classic tools such as EvoSuite [14] for Java and Pynguin [15] for Python search for inputs that raise coverage, but the tests they produce often have unnatural names and weak assertions. LLMs trained on code can write tests that look like a developer wrote them, which is why the idea spread quickly after 2022.

The difficulty for a newcomer is that every paper measures something slightly different. One paper reports statement coverage on JavaScript packages, another reports the share of generated tests that compile, and an industrial paper reports how many suggestions engineers accepted. A reader who just compares the biggest numbers will draw the wrong conclusion. This survey tries to put the evidence in one place and to be explicit about what can and cannot be compared.

The topic also moves fast. Figure 1 shows when the venue versions of the studies reviewed here appeared. Most were published within about three years, which is one reason the evaluation practices have not settled.



Figure 1. Timeline of the studies reviewed in this survey, placed by the approximate time of their journal or conference publication.

1.1 Related Surveys And What Is Different Here

Several surveys already exist. Wang et al. analyse 102 papers on LLMs across software testing in general [1]. Perera and De Silva review the metrics used to evaluate LLM-based test generation [2], and Zapkus and Slotkienė give an earlier review of LLMs for unit test generation [13]. Because these exist, this paper does not try to be the broadest survey. Its contribution is narrower: a compact side-by-side view of representative approaches, with attention to how each one is evaluated, and a practical checklist that a student or small team can apply when reading or writing a paper in this area.

1.2 Research Questions

- RQ1. What kinds of approaches are used to generate unit tests with LLMs?

© 2026 IJSRET

- RQ2. How are these approaches evaluated, and can the reported results be compared?
- RQ3. What gaps remain, and what should future studies report?

1.3 Contributions

- A five-family taxonomy of LLM-based unit test generation, with representative studies in each family.
- A comparison table that records the setting, technique and reported result of each study, kept separate from the author's own interpretation.
- An analysis of evaluation practice: which metrics are used, which benchmarks exist, and where the evidence is thin.
- A reporting checklist and a set of research directions for future work. Both are proposals and have not been tested empirically.

1.4 Organisation Of The Paper

Section 2 describes how studies were selected. Section 3 gives the background on unit tests, metrics and traditional generators. Section 4 presents the taxonomy. Section 5 reviews each study in turn. Section 6 analyses evaluation practice and Section 7 answers the research questions. Section 8 lists challenges and threats to validity, Section 9 proposes the reporting checklist, Section 10 outlines future directions, and Section 11 concludes. Appendices list abbreviations and the extraction template.

II. SURVEY METHOD

This is a focused literature survey, not a systematic review, and it does not claim to follow PRISMA. The aim was to cover the main technique families with well-documented, representative studies rather than to count every paper.

2.1 Search

Papers were located in October and November 2026 using web search and the pages of publishers and conferences. Queries were built from terms such as "LLM unit test generation", "ChatGPT unit tests", "coverage-guided LLM test generation" and "unit test generation benchmark". Only studies with a formal journal or conference venue were kept. Earlier surveys [1][2][13] were also read for the lists of studies they cover, which helped to check that no major technique family was missed.

2.2 Inclusion And Exclusion

Table 1. Selection criteria

Type	Criterion	Reason
Include	An LLM is used to generate or improve unit tests	Keeps the focus on LLM-based unit testing
Include	The paper reports an empirical result (coverage, pass rate, acceptance, etc.)	Allows results to be recorded
Include	Published in a journal or conference, 2023 or later	Covers the period after LLMs became widely used for code
Include	Benchmark papers that evaluate LLM test generation	Needed to discuss evaluation practice
Exclude	Fuzzing, GUI testing, test repair, and non-LLM neural models	Different problem settings
Exclude	Papers with no accessible abstract or text	Results could not be recorded reliably

2.3 Data Extraction

For each paper the author recorded the venue and year, the programming language and subject programs, the technique, the evaluation metrics, and the headline result (Appendix B lists the template). Numbers in this survey are copied from the papers' abstracts or text and are attributed to the authors; they were not reproduced. Where a detail could not be found in the material consulted, it is left blank or marked with a dash rather than guessed.

2.4 Limits Of The Method

Because selection was purposive, relevant work may be missing. The extraction was done by one person, so there was no second reviewer to check for errors. These limits are discussed again in Section 8.2.

III. BACKGROUND

3.1 Unit Tests And What Makes A Good One

A unit test sets up an input, runs a small piece of code and checks the output with an assertion. Two parts of this are hard to automate. The first is building a realistic input, which may require creating objects, mocking dependencies or reading documentation. The second is the test oracle: deciding what the correct output should be. If a generator writes the expected value by running the current code, the test will pass but may simply record a bug. This is why many of the approaches reviewed here describe their output as regression

© 2026 IJSRET

tests, which assume the code under test is currently correct [7][10].

3.2 A Ladder Of Quality Measures

A generated unit test can be judged at five levels, shown in Figure 2. The first level is whether it compiles and runs. The second is whether it passes on the current code. The third is how much code it covers, usually measured as line or branch coverage. The fourth is whether it would actually catch a bug; mutation score, the share of artificially injected faults that a test suite detects, is the usual proxy, and Just et al. found mutants to be a valid substitute for real faults in their study [16]. The fifth is whether developers find the test readable and worth keeping.

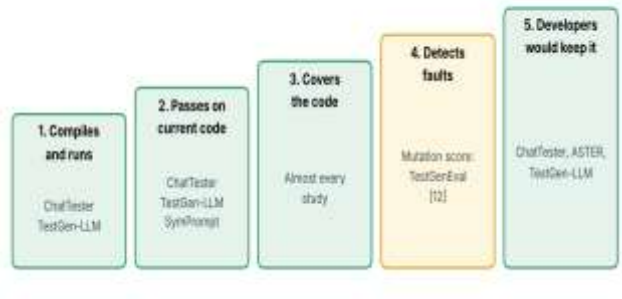


Figure 2. Five levels at which generated tests can be judged, with the reviewed studies that report each level (as found in the text consulted). Higher levels are reported less often.

Climbing this ladder gets progressively more expensive. Compilation and coverage are cheap to measure automatically. Mutation analysis needs many extra test runs. Developer judgement needs people. This cost gradient explains much of the pattern seen later in the review, where the cheap measures dominate.

3.3 Traditional Generators

Search-based tools such as EvoSuite [14] and Pynguin [15] evolve test inputs to raise coverage. Their weak spot is that they struggle to build realistic objects and meaningful assertions, and the resulting tests are often hard to read. LLM-based work tries to fill this gap using what the model has learned about how code is normally used. A recurring finding, shown by CodaMosa, is that the two approaches combine well [6].

3.4 How An LLM Is Used To Write Tests

In the simplest setting the model is given the source of the function under test and asked for a test. Richer settings add the signature of related functions, usage examples from documentation, the existing test file or the coverage report. Three choices matter in practice. The first is which model is used: the studies here range from Codex and CodeGen2 to GPT-3.5, GPT-4 and models trained inside a company. The second is how much context fits in the prompt, which is why ChatUniTest builds an adaptive focal context within a token limit [5]. The third is what happens after generation, since a test that fails to compile is of no use. The families in Section 4 differ mainly in how they answer these three questions.

IV. TAXONOMY OF APPROACHES (RQ1)

The reviewed studies fall into five families, shown in Figure 3. The families overlap, since a hybrid tool may also repair failing tests, but each has a distinct main idea. This taxonomy is the author's own organisation of the studies, and other ways of grouping them are possible.

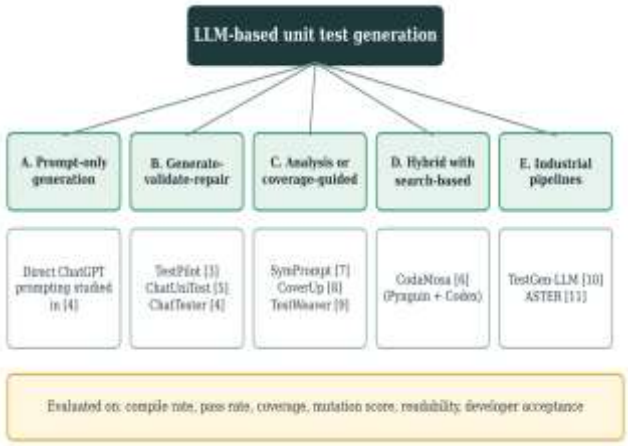


Figure 3. Taxonomy of the LLM-based unit test generation approaches reviewed in this paper. Bracketed numbers refer to the reference list.

A. Prompt-only generation. The model receives the function under test and writes tests in one shot. The empirical study behind ChatTester examined exactly this for ChatGPT and found that only 24.8% of tests passed when executed [4]. This family is mainly useful as a baseline.

B. Generate-validate-repair loops. The tool generates a test, compiles and runs it, and sends compiler or runtime errors back to the model (Figure 4). TestPilot re-prompts the model with the failing test and its error message [3]. ChatUniTest builds an adaptive focal context and then uses rule-based repair followed by model-based repair [5]. ChatTester adds an initial generator and an iterative refiner [4].

© 2026 IJSRET

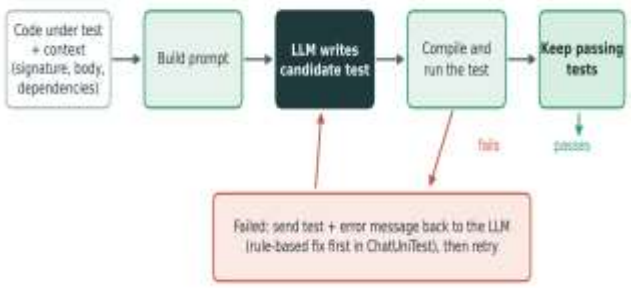


Figure 4. Generic generate-validate-repair loop used by the second family. The details differ by tool; this diagram is a simplified summary drawn for this survey.

C. Analysis or coverage-guided generation. Program analysis tells the model what to target. SymPrompt turns execution paths of the method under test into a sequence of prompts [7]. CoverUp measures coverage and puts the uncovered lines into the next prompt [8]. TestWeaver adds execution-aware feedback for regression test generation [9].

D. Hybrids with search-based testing. CodaMosa runs the Pynguin search and calls the LLM only when coverage stops improving, then seeds the search with the model's tests (Figure 5) [6].

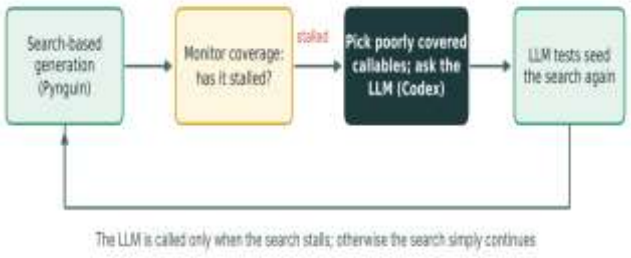


Figure 5. Hybrid approach of CodaMosa [6], simplified from the description in the paper.

E. Industrial pipelines. TestGen-LLM at Meta improves existing Kotlin test classes and keeps a candidate only if it builds, passes reliably and adds coverage [10]. ASTER uses static analysis to guide generation for Java and Python and studies how natural the resulting tests look to developers [11].

V. REVIEW OF SELECTED STUDIES

This section describes each study in turn, using the authors' own reported figures. Table 2 then summarises them. The results come from different languages, models and benchmarks, so the last column of the table must not be read as a ranking.

5.1 Testpilot

TestPilot generates tests for JavaScript and was evaluated on 25 npm packages with 1,684 API functions [3]. The prompt contains the signature and implementation of the function under test, and it can include usage examples taken from documentation. If a generated test fails, TestPilot builds a new prompt that includes the failing test and its error message and asks the model to fix it. The authors report that with gpt3.5-turbo the generated tests reached a median statement coverage of 70.2% and a median branch coverage of 52.8%, against 51.3% and 25.6% for Nessie, an earlier feedback-directed technique [3]. They also tried two other models, code-cushman-002 and the open StarCoder, and report that LLM-based generation outperformed prior methods on key metrics [3].

5.2 Chattester

Yuan et al. start with an empirical study of ChatGPT for unit test generation [4]. Their findings are mixed. Only 24.8% of the tests ChatGPT produced passed; 57.9% had compilation errors and 17.3% compiled but failed, mostly because of wrong assertions. On the other hand, the passing tests resembled hand-written ones in coverage and readability and were sometimes preferred by developers. Based on this, they propose ChatTester, which has an initial test generator and an iterative test refiner, and report 34.3% more compilable tests and 18.7% more tests with correct assertions than default ChatGPT. The approach was also applied to CodeLlama-Instruct and CodeFuse and improved their tests, which suggests it is not tied to one commercial model [4].

5.3 Chatunitest

ChatUniTest is a Java tool built on the generation-validation-repair idea [5]. Instead of sending the whole project to the model, it parses the project and builds an adaptive focal context: the focal method and its dependencies, trimmed to fit a token limit. The generated test is then validated by compiling and running it. Repair comes in two steps. Rule-based repair handles simple problems such as syntax errors and missing imports, and a model-based repair step sends harder errors back to ChatGPT together with the broken test and its context. The authors package the approach as a library and a Maven plugin, and report the best overall line coverage among the tools compared, ahead of TestSpark and EvoSuite in half of the evaluated projects [5].

5.4 Codamosa

CodaMosa targets a known weakness of search-based testing: coverage plateaus, where the search stops making progress [6]. It runs the Pynguin search and monitors

© 2026 IJSRET

coverage. When progress stalls, it finds callables that are poorly covered and asks Codex to write tests for them, then feeds those tests back into the search. The authors evaluated it on 486 Python modules and report significantly higher coverage on many more benchmarks than the baselines (173 and 279 modules in the two comparisons reported) [6]. The study matters because it shows the LLM need not replace the search tool; it can be called selectively, which also limits the cost.

5.5 Symprompt

SymPrompt, also called code-aware prompting, works in the regression setting, where the code is assumed correct and tests are written to detect later changes [7]. Its idea is to break test-suite generation into a sequence of prompts, each aligned with one execution path of the method under test, using static analysis to derive the path constraints. The authors report that for CodeGen2 it raises the number of correct generations by a factor of 5 and relative coverage by 26%, and that with GPT-4 it more than doubles coverage compared with baseline prompts [7].

5.6 Coverup

CoverUp drives generation with coverage data [8]. It measures the coverage of the current test suite, then writes prompts that point the model at the lines and branches that are still uncovered, and it repeats the process as coverage changes. The authors compare it with two state-of-the-art systems, CodaMosa and MuTAP, a mutation-and-LLM approach. Against CodaMosa they report a per-module median line and branch coverage of 80% versus 47%; against MuTAP, using GPT-4o, an overall line and branch coverage of 90% versus 77%. They also show that its performance comes from the combination of its components and not only from the LLM used [8].

5.7 Testweaver

TestWeaver, published at ICSE 2026, is a coverage-guided approach for regression test generation that adds execution awareness [9]. The authors note that existing LLM-based methods often stall in what they call a coverage plateau, because the model reasons poorly about how the program actually executes. TestWeaver gives the model three kinds of help: the backward slice from the target line instead of the full program, the existing test whose execution path is closest to the uncovered line, and inline comments showing variable values along the executed path. The authors report that this accelerates coverage growth and produces more effective tests than the state of the art [9]. The text consulted did not give enough detail to record specific numbers, so none are quoted here.

5.8 Testgen-Llm At Meta

TestGen-LLM is an industrial tool, and its design differs from the others because it improves existing human-written Kotlin test classes rather than writing tests from nothing [10]. It uses two of Meta's own LLMs and applies a series of filters to every candidate, so that only tests that build, pass reliably and increase coverage are recommended. The authors call this "Assured Offline LLMSE". Because the tool cannot decide automatically whether a failing test has found a real bug, it simply discards candidates that fail, and repeated execution is used to remove flaky tests [10]. In an evaluation on Instagram's Reels and Stories products, 75% of the test cases built correctly, 57% passed reliably and 25% increased coverage (Figure 6). In Meta's test-a-thons on Instagram and Facebook, it improved 11.5% of the classes it was applied to, and engineers accepted 73% of its recommendations for production [10].

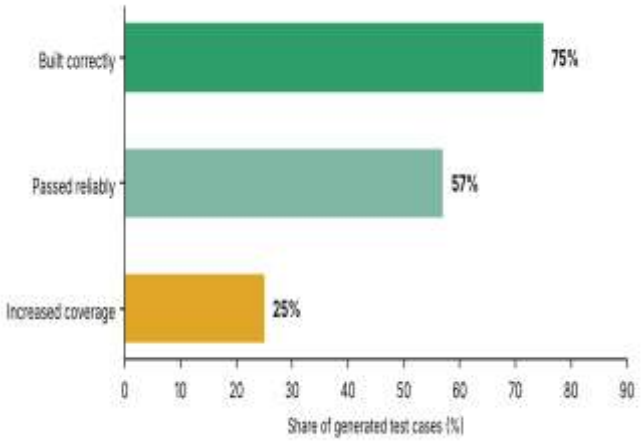


Figure 6. Share of TestGen-LLM test cases that passed each filter in the Instagram Reels and Stories evaluation, as reported in [10].

5.9 Aster

ASTER is a multi-language generator that uses static analysis to guide the LLM and then checks the generated tests for compilation and runtime errors, building new prompts to repair them [11]. It was evaluated on standard and enterprise Java applications and on a large Python benchmark, and it measures both coverage and how natural the tests look. The authors report that it is competitive with, and sometimes better than, state-of-the-art techniques on coverage and produces considerably more natural tests. They also report a user study with 161 professional developers on the naturalness of the generated tests [11]. ASTER is one of the few studies in this set that tests more than one language and measures human perception directly.

© 2026 IJSRET

5.10 Summary Table

Table 2. Reviewed approaches and their reported results

Study	Venue, year	Setting	Technique	Result as reported by the authors
TestPilot [3]	IEEE TSE, 2024	JavaScript; 25 npm packages, 1,684 API functions	Prompt with signature, body and doc examples; re-prompt on failure	gpt3.5-turbo: median statement coverage 70.2%, branch 52.8%; Nessie baseline 51.3% and 25.6%
ChatTester [4]	FSE, 2024	ChatGPT; also CodeLlama-Instruct and CodeFuse	Initial test generator plus iterative refiner	Default ChatGPT: 24.8% passing. ChatTester gives 34.3% more compilable and 18.7% more correct-assertion tests
ChatUniTest [5]	FSE Companion, 2024	Java projects, Maven plugin	Adaptive focal context; generation-validation-repair	Best overall line coverage, ahead of TestSpark and EvoSuite in half the projects
CodaMosa [6]	ICSE, 2023	Python; 486 modules	Pynguin search; Codex called when coverage stalls	Significantly higher coverage on many more benchmarks than baselines (173 and 279 modules)
SymPrompt [7]	FSE, 2024	Python methods from open-source projects	Path-constraint prompts from static analysis	CodeGen2: 5x more correct generations, +26% relative coverage. GPT-4: over 2x coverage vs baseline prompts
CoverUp [8]	FSE, 2025	Python	Coverage-guided iterative prompting	Per-module median line+branch coverage 80% vs 47% (CodaMosa); overall 90% vs 77% (MuTAP, GPT-4o)
TestWeaver [9]	ICSE, 2026	Regression test generation	Backward slice, closest test and execution annotations	Authors report faster coverage growth and more effective tests than the state of the art
TestGen-LLM [10]	FSE Companion, 2024 (industry)	Kotlin; Instagram and Facebook	Extends existing tests; filters for build, reliable pass, coverage gain	75% built, 57% passed reliably, 25% raised coverage; 11.5% of classes improved; 73% of recommendations accepted
ASTER [11]	ICSE-SEIP, 2025	Java (incl. enterprise apps) and Python	Static-analysis-guided generation and repair	Competitive with or better than state of the art on coverage; more natural tests; survey of 161 professional developers

Two of the studies compare against named baselines on the same subjects, so their numbers can be set side by side within the study. Figure 7 does this for TestPilot against Nessie and CoverUp against CodaMosa and MuTAP. The figure does not compare the two studies with each other, because they use different languages and subject programs.

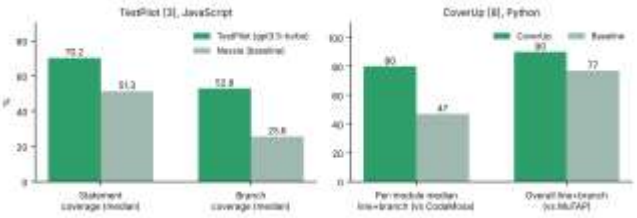


Figure 7. Within-study comparisons reported by the authors: TestPilot against Nessie [3] (left) and CoverUp against CodaMosa and MuTAP [8] (right). Values are medians or overall figures as stated in each paper.

5.11 Observations

Three observations follow. First, repair loops and analysis guidance appear in almost every recent approach, which suggests that raw prompting is no longer considered enough. Second, most work targets Python, Java or JavaScript, with Kotlin appearing only in an industrial setting. Third, the only study that reports acceptance by working engineers is the industrial one [10], and the ASTER developer survey is the other place where human judgement is measured directly [11].

5.12 How Often Do Generated Tests Simply Fail?

The clearest picture of the correctness problem comes from the ChatTester study. Of the tests default ChatGPT produced, 57.9% hit compilation errors such as undefined symbols or

© 2026 IJSRET

private-field access, 17.3% compiled but failed at run time, mostly because of wrong assertions, and 24.8% passed [4].

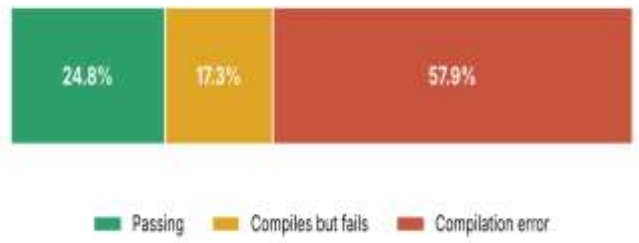


Figure 8. Outcome of tests generated by default ChatGPT in the ChatTester study [4]. Values are as reported by the authors.

Figure 8 shows the split. The same paper also reports that the passing tests were comparable to hand-written ones in coverage and readability and were sometimes preferred by developers, which is why repair loops are attractive: the model's output is useful once the broken part is fixed.

VI. EVALUATION PRACTICES (RQ2)

6.1 Metrics Used

Table 3. Metrics seen in the reviewed studies

Study	Metrics reported (from abstract or text consulted)	Human or developer judgement
TestPilot [3]	Statement and branch coverage	—
ChatTester [4]	Compilation rate; passing rate; coverage; readability	Developer preference study
ChatUniTest [5]	Line, branch and focal-method coverage	—
CodaMosa [6]	Coverage on 486 modules	—
SymPrompt [7]	Share of correct generations; relative coverage	—
CoverUp [8]	Per-module and overall line+branch coverage	—
TestGen-LLM [10]	Build rate; reliable-pass rate; coverage increase	Engineer acceptance of recommendations
ASTER [11]	Coverage; naturalness of tests	Survey of 161 developers
TestGenEval [12]	pass@k; coverage; mutation score	—

Coverage appears in nearly every study. Mutation score does not, apart from the benchmark paper that makes it a core metric. The TestGenEval authors observe that existing benchmarks mostly report pass@k, a few report coverage and none report mutation score, even though mutation score correlates best with real fault detection [12]. Coverage alone is a weak guide: a test can execute a line without checking anything about its result. Few of the studies consulted discuss memorisation, even though test code is widely available online and may appear in a model's training data. This is a risk worth reporting on, although this survey could not confirm which papers examined it.

6.2 Benchmarks

Early studies tended to use small, self-contained functions. TestGenEval moves to realistic code: it contains 68,647 tests from 1,210 code and test file pairs across 11 Python repositories, built from SWE-bench, and it covers both full test-suite generation and test completion [12]. On this benchmark the best model, GPT-4o, reached an average coverage of only 35.2% [12]. The gap between that figure and the 70.2% median reported by TestPilot on npm packages [3] should not be read as a contradiction, because the languages, projects, models and tasks all differ. It does show how strongly the benchmark chosen shapes the headline number.

6.3 Why Results Cannot Be Ranked

- Different languages and programs: JavaScript, Python, Java and Kotlin results are not interchangeable.
- Different models: results range from Codex and CodeGen2 to GPT-4 and in-house models, and model versions change over time.
- Different metrics: compile rate, pass rate, coverage and acceptance measure different things.
- Different baselines: some papers compare against EvoSuite, some against Pynguin, CodaMosa or MuTAP, and some against a plain prompt.
- Different units of reporting: medians per module, overall coverage and averages over projects are not the same statistic.

For this reason Table 2 is a record of what each paper claims, not a leaderboard.

VII. ANSWERS TO THE RESEARCH QUESTIONS

RQ1: approaches. The studies fall into five families, from prompt-only generation to industrial pipelines (Section 4). The most active direction is the combination of an LLM with something that checks it: a compiler and test runner in repair

© 2026 IJSRET

loops, a coverage tool in coverage-guided approaches, or a search algorithm in hybrids.

RQ2: evaluation and comparability. Evaluation leans on cheap automatic measures, mainly coverage. Mutation score and human judgement appear in few studies, and benchmark choice changes the headline numbers a great deal (Section 6). Results from different papers therefore cannot be ranked, though within-paper comparisons against a named baseline are informative.

RQ3: gaps. The main gaps are inconsistent metrics, missing mutation analysis, limited human evidence, narrow language coverage, and reproducibility problems caused by changing commercial models. Section 9 turns these into a reporting checklist.

VIII. CHALLENGES AND THREATS

8.1 Challenges In The Field

Table 4. Recurring challenges and the evidence for them in the reviewed papers

Challenge	Evidence in the reviewed studies	Common response
Compilation and hallucinated symbols	57.9% of default ChatGPT tests had compilation errors [4]	Validate and repair loops [3][4][5]
Wrong assertions (oracle problem)	17.3% compiled but failed, mostly from incorrect assertions [4]; Meta cannot decide automatically whether a failing test found a bug, so it keeps only tests that pass [10]	Regression framing; discard failures
Low coverage on real projects	Best model reached 35.2% average coverage on TestGenEval [12]	Coverage-guided prompting [7][8]
Flaky tests	TestGen-LLM filters candidates by repeated execution to reject flaky ones [10]	Repeated runs before acceptance
Memorisation of training data	Rarely discussed in the papers consulted; test code is widely available online	Use projects newer than the model; similarity checks
Language and domain bias	Evidence concentrates on Python, Java and JavaScript (Table 2)	More languages in benchmarks
Reproducibility	Several studies use commercial models whose versions change; model names alone do not identify a version	Report model version, date and settings

The oracle problem deserves a closer look because it limits what a generated test can mean. A test that passes only shows that the code still behaves as it did when the test was written. It does not show that the behaviour is right. The industrial pipeline at Meta accepts this openly and frames its tests as protection against regressions, not as bug finders [10]. Papers that claim bug-finding ability need to show it with injected or real faults, which is where mutation analysis helps.

8.2 Threats To Validity Of This Survey

- Selection bias. The selection is purposive, so the survey is not exhaustive and may favour well-publicised work. Restricting the survey to papers with a formal venue also leaves out recent work that has not yet been published.
- Version differences. Numbers may differ between the conference and journal versions of a paper, and the figures here follow the versions consulted.
- Extraction error. Results were taken from abstracts and the text available to the author, by one person, and were not reproduced.
- Taxonomy subjectivity. The five families are the author's own organisation of the literature.
- Recency. The field is moving quickly, and newer work may already change some of the conclusions.

Readers should check any number against the original paper before relying on it.

IX. PROPOSED REPORTING CHECKLIST

The checklist below follows from the gaps found in Sections 5 to 8. It is a proposal for future studies and has not been validated.

Table 5. Gaps and proposed actions

Gap	Proposed action	What to report
Metrics differ across papers	Report compile rate, pass rate, coverage and mutation score together	All four, with the tool used to measure each
Coverage can hide weak assertions	Add mutation score on at least a sample of projects [12][16]	Mutation tool, mutants generated, score
Model versions drift	Record exact model, version, date and temperature	Model ID, access date, prompt template

© 2026 IJSRET

Gap	Proposed action	What to report
Possible training-data overlap	Include projects created after the model's training cutoff or run similarity checks	Cutoff date, similarity method
Little human evidence	Add a small developer study on readability and acceptance [4][10][11]	Participants, tasks, rating scale
Narrow language coverage	Evaluate on at least one language beyond Python and Java	Language, framework, subject programs
Cost is rarely reported	Report tokens, time and number of repair rounds per test	Mean and spread per method

9.1 Questions A Reader Can Ask Of Any Paper

- Which model and version produced the results, and when was it accessed?
- Were the subject programs likely to be in the model's training data?
- Is the coverage figure a median per module, an overall figure or an average over projects?
- Is there a baseline, and does it run on exactly the same subjects?
- Were tests checked for flakiness by running them more than once?
- Did anyone other than the authors judge the quality of the tests?

X. FUTURE RESEARCH DIRECTIONS

Shared benchmarks with mutation scoring. TestGenEval is a step toward realistic evaluation [12], but it covers Python only. Similar benchmarks for Java, JavaScript and other languages, with mutation scores reported by default, would make results comparable.

Reproduction studies. Re-running two or three approaches on the same subjects, with the same model and the same metrics, would show how much of the reported gap between tools survives a fair comparison. This is the natural follow-up to this survey.

Better oracles. Since a generated assertion may only record current behaviour, work on checking expected values against documentation or specifications could make tests more useful for finding bugs and not only for regression protection.

Cost-aware evaluation. Few studies report tokens, time or repair rounds, yet these decide whether a tool is practical. Reporting cost next to coverage would give a fairer picture of the trade-off.

Human studies at scale. Industrial reports such as Meta's [10] and developer surveys such as ASTER's [11] suggest that acceptance by developers is measurable. More such studies, especially outside large companies, would show whether the tests are used in practice.

Open models. Several results depend on commercial models that change without notice. Evaluating the same techniques with open models would improve repeatability, and ChatTester's results on CodeLlama-Instruct and CodeFuse are an early sign that this is feasible [4].

XI. CONCLUSION

LLM-based unit test generation has moved quickly from asking a model for a test to building pipelines that generate, run, repair and filter tests. The studies reviewed here agree that validation and analysis guidance matter, because raw model output often fails: in one study only 24.8% of default ChatGPT tests passed [4]. They disagree, or at least cannot be compared, on how good the results are, since languages, models, benchmarks and metrics differ. Coverage dominates the evidence, mutation score and human judgement are scarce, and realistic benchmarks such as TestGenEval show that coverage on real repositories is still modest [12]. The most useful near-term improvement is therefore not a new tool but more consistent reporting. The checklist in Section 9 is offered as a starting point, and testing it on a small set of reproduced experiments is the natural next step.

1 Appendix A. Abbreviations

Term	Meaning
LLM	Large language model
SBST	Search-based software testing
CUT / focal method	The code or method for which a test is generated
Oracle	The rule or expected value that decides whether a test result is correct
Line / branch coverage	Share of lines, or of decision outcomes, executed by the tests
Mutation score	Share of injected faults (mutants) that a test suite detects
pass@k	Probability that at least one of k generated samples passes
Regression test	A test written to detect later changes in behaviour, assuming the current code is correct
Flaky test	A test that sometimes passes and sometimes fails without any change in the code

© 2026 IJSRET

12.1 Appendix B. Data Extraction Template

The following fields were recorded for each study. A blank field means the information was not found in the material consulted.

Field	What was recorded
Citation and venue	Authors, title, venue, year, DOI
Language and subjects	Programming language, number and type of projects or functions
Model	LLM names and sizes, and whether commercial or open
Technique	Prompting method, validation or repair steps, guidance from analysis
Baselines	Tools or prompts the approach was compared against
Metrics	Compile or pass rate, coverage, mutation score, human evaluation
Headline result	The main numbers, copied as stated by the authors
Stated limits	Limitations or threats the authors mention

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 Transactions on Software Engineering, 2024, doi: 10.1109/TSE.2024.3368208.
2. V. I. T. Perera and D. I. De Silva, “Evaluating Large Language Models for Software Testing: A Systematic Review of Metrics and Practices,” 2026 (institutional repository record, SLIIT).
3. M. Schäfer, S. Nadi, A. Eghbali and F. Tip, “An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation,” IEEE Transactions on Software Engineering, vol. 50, no. 1, pp. 85–105, 2024.
4. Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen and X. Peng, “Evaluating and Improving ChatGPT for Unit Test Generation,” Proc. ACM on Software Engineering (FSE), 2024.
5. Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng and J. Yin, “ChatUniTest: A Framework for LLM-Based Test Generation,” in Companion Proc. ACM Int. Conf. on the Foundations of Software Engineering (FSE Companion), 2024.
6. C. Lemieux, J. P. Inala, S. K. Lahiri and S. Sen, “CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models,” in Proc. IEEE/ACM 45th Int. Conf. on Software Engineering (ICSE), 2023, pp. 919–931, doi: 10.1109/ICSE48619.2023.00085.
7. G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan and B. Ray, “Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM,” Proc. ACM on Software Engineering (FSE), 2024.
8. J. Altmayer Pizzorno and E. D. Berger, “CoverUp: Effective High Coverage Test Generation for Python,” Proc. ACM on Software Engineering (FSE), 2025, doi: 10.1145/3729398.
9. C. C. Le, C. D. Van, T. D. Vu, M. V. T. Pham, H. N. Phan, H. N. Phan and T. N. Nguyen, “TestWeaver: Execution-aware, Feedback-driven Regression Testing Generation with Large Language Models,” in Proc. IEEE/ACM 48th Int. Conf. on Software Engineering (ICSE), Rio de Janeiro, 2026, doi: 10.1145/3744916.3787805.
10. N. Alshahwan, J. Chheda, A. Finegenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta and E. Wang, “Automated Unit Test Improvement using Large Language Models at Meta,” in Companion Proc. ACM FSE 2024, pp. 185–196, doi: 10.1145/3663529.3663839.
11. R. Pan, M. Kim, R. Krishna, R. Pavuluri and S. Sinha, “ASTER: Natural and Multi-language Unit Test Generation with LLMs,” in Proc. IEEE/ACM Int. Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2025.
12. K. Jain, G. Synnaeve and B. Rozière, “TestGenEval: A Real World Unit Test Generation and Test Completion Benchmark,” in Proc. Int. Conf. on Learning Representations (ICLR), 2025.
13. D. M. Zapkus and A. Slotkienė, “Unit Test Generation Using Large Language Models: A Systematic Literature Review,” Lietuvos magistrantų informatikos ir IT tyrimai, 2024, pp. 136–144, doi: 10.15388/LMITT.2024.20.
14. G. Fraser and A. Arcuri, “EvoSuite: Automatic Test Suite Generation for Object-Oriented Software,” in Proc. ESEC/FSE, 2011, pp. 416–419.
15. S. Lukasczyk and G. Fraser, “Pynguin: Automated Unit Test Generation for Python,” in Proc. ICSE Companion, 2022, pp. 168–172, doi: 10.1145/3510454.3516829.
16. R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes and G. Fraser, “Are Mutants a Valid Substitute for Real Faults in Software Testing?,” in Proc. ACM SIGSOFT FSE, 2014, pp. 654–665.

© 2026 IJSRET