Software Quality Journal (2026) 34:59
https://doi.org/10.1007/s11219-026-09790-3

RESEARCH

LLM-driven test and test-repair automation for legacy Perl systems: an empirical study at IBM

Simon Leunig[1] · **Kai Petersen**[2,3]

Received: 29 March 2026 / Accepted: 19 September 2026
© The Author(s) 2026

Abstract
Automated unit test generation using Large Language Models (LLMs) has shown promise for widely used programming languages such as Java and Python. It may further benefit legacy systems lacking a test base, and the investigation of less widespread legacy languages remains relevant as many companies still operate systems built on them. We develop a modular workflow combining static code analysis, multi-stage prompt engineering with business logic extraction, automated error analysis, and iterative test-fix cycles for legacy systems, evaluated by pass rates, coverage, and repair effectiveness. We evaluate our approach on four productive IBM Perl repositories (186 functions) and three CPAN distributions (139 functions) using two open-source LLMs: Llama-3.3-70B-Instruct and Mistral-Small-3.1-24B-Instruct-2503. A key methodological contribution is a defensive `eval` wrapper that records a crash in the code under test as a failed assertion rather than terminating the run. A git-diff mode enables selective test generation for changed functions, supporting incremental maintenance workflows. The larger model achieves initial pass rates of 66–82% (industrial) and 74–81% (CPAN), improving to 74–86% and 76–89% after repair, with statement coverage of 34–92% and 65–100%. The smaller model performs substantially worse on both sets. Repair reduces structural errors on both (`no_output` from 0–44% to 0–6% and 0–7% to 0–5%) and improves pass rates for both models in most repositories. We show the transferability of LLM-based test generation to low-resource legacy languages, establishing Perl as a viable target for automated testing pipelines.

Keywords Large language models · Automated test generation · Perl · Legacy code · Software testing · CI/CD integration

✉ Simon Leunig
leunigs@hsu-hh.de

Kai Petersen
kai.petersen@hs-flensburg.de; kai.petersen@bth.se

[1] Helmut Schmidt University Hamburg, Hamburg, Germany

[2] University of Applied Sciences Flensburg, Flensburg, Germany

[3] Blekinge Institute of Technology, Karlskrona, Sweden

Published online: 21 October 2026

1 Introduction

Software testing is a prerequisite for ensuring quality, correctness, and reliability of software products (cf. Wang et al., 2024a). In particular, unit tests allow for the isolated verification of individual functions and help to detect errors early and limit the impact of changes (Tang et al., 2024). However, many existing systems lack sufficient test coverage. The manual creation of unit tests is time-consuming, repetitive, and resource-intensive, leading to gaps in test coverage (Zhang et al., 2025). Traditional automation tools such as Search-Based Software Testing (SBST) often achieve good coverage values, but deliver tests that are limited in terms of readability and maintainability (Schäfer et al., 2024). Against this background, the use of generative AI (GenAI) and large language models (LLMs) is gaining importance: these models can generate human-like readable tests and have the potential to qualitatively improve the automation of test generation (Yuan et al., 2024).

At the same time, there are challenges: LLMs occasionally generate syntactically correct but semantically incorrect code (so-called hallucinations) (Walczak et al., 2026), their results are probabilistic and reproducibility can be limited (Jalil et al., 2023), and the oracle problem (Aleti, 2023) complicates the objective evaluation of test quality. Moreover, previous research has focused mainly on popular languages such as Java or JavaScript; for Perl (Wall & Loukides, 2000) no study exists in this context, which represents a concrete research gap that this work addresses. Perl is often referred to as a low-resource programming language, which means that there is limited training data available, in the LLM DeepSeek-Coder, for example, the proportion for Perl is only 0.10% compared to Java with 18.63% (Guo et al., 2024; Wong et al., 2025). Furthermore, according to Kotti et al. (2025), Perl universally exhibits high perplexity, which means that AI models show significantly greater uncertainty when predicting this code than with other languages such as Java (however, this depends on the LLM used).

The main objective (MO) of this work is the design and implementation of a modular workflow for the automated generation of unit tests for Perl code using LLMs, including a Test-Fix-cycle. That is, the system analyzes the observed failures and fixes them. Thereafter, a retest is run.

The empirical investigation has the following objectives:

- Objective 1 (O1): Assess the ability to generate syntactically correct and executable unit tests for legacy Perl modules.
- Objective 2 (O2): Investigate the ability of our solution to repair the problems identified.
- Objective 3 (O3): Investigate the code coverage achieved after unit test generation and repair.

For each of the objectives above, we compare different LLM models. The focus in this research is on local models, as the studied company (as well as other companies) strives for digital sovereignty. That is, in the context of this research it was not possible to use cloud-based models.

We evaluate the approach on four productive IBM Perl repositories comprising 186 functions, and on three publicly available distributions from the Comprehensive Perl Archive Network (CPAN) comprising 139 functions, comparing two open-source LLMs of differing scale: Llama-3.3-70B-Instruct and Mistral-Small-3.1-24B-Instruct-2503.

For O1 and O2, we assess the syntactic and semantic correctness of generated tests through pass rates and the distribution of error types, both before and after test repair. Failed tests are classified into three categories: `no_output` (structural errors where the test file cannot be executed at all), `test_failure` (tests that execute but fail assertions), and `unknown` (failures that cannot be unambiguously classified into either category). For O3, we measure code coverage (statement, branch, condition, and subroutine) using `Devel::Cover`, comparing results across both models.

The structure of the work is organized as follows: Section 2 (Related Work) presents the relevant concepts of software testing, the functioning of LLMs, and the current state of research. Section 3 (Solution Description) describes the objectives, design, and overall workflow of the proposed solution and the technical details for implementation, the tools used, and the prompt engineering approach. Section 4 (Evaluation) shows the evaluation, including the methodology and results. Section 5 (Discussion) interprets the results, discusses limitations, and provides recommendations for action. Section 6 (Conclusion) summarizes the key findings and provides an outlook on further research.

2 Related work

To understand the use of generative artificial intelligence (AI) for automating test case creation, it is important to show the fundamental concepts of software testing, the functioning of Large Language Models (LLMs), and the specific challenges that arise at the intersection of these two areas.

2.1 Test automation

To reduce these manual efforts for test design and execution, various approaches to automated test case generation have been developed. Schäfer et al. (2024) discusses in this context various approaches to automated test generation, including feedback-directed random test generation, dynamic symbolic execution, fuzzing, and search- and evolution-based methods. In particular, traditional techniques such as Search-Based Software Testing (SBST), for which tools like EvoSuite are prominent representatives, aim to achieve the highest possible code coverage through evolutionary algorithms (Tang et al., 2024). Although these methods often deliver good results in terms of coverage, the generated tests have weaknesses: they are often difficult to read and understand, contain meaningless variable names, and do not reflect human intention, which limits their acceptance and maintainability in practice (cf. Schäfer et al., 2024).

2.2 Generative AI and Large Language Models (LLMs)

Generative AI (GenAI) and Large Language Models (LLMs) represent a shift, as they not only analyze existing data but generate new, original content (Bajaj & Samal, 2023).

For specific tasks such as code generation or the creation of unit tests, these base models are used in two primary ways as described in Wang et al. (2024a):

1. Prompt Engineering: Through carefully designed instructions (prompts), the knowledge present in the model is made usable for a specific task without changing the model itself. Techniques such as Zero-Shot-Learning (direct instruction), Few-Shot-Learning (instruction with examples), or Chain-of-Thought (CoT), where the model is guided to formulate its ”chain of thought” step by step, are intended to improve the quality of the generated results.
2. Fine-Tuning: Here, a pre-trained model is retrained with a smaller, task-specific dataset to adjust its weights and specialize it for a particular domain.

Through these capabilities, LLMs are able to generate human-like and contextually appropriate code, making them a good candidate for overcoming the readability problems of traditional test generation tools.

2.3 Specific challenges in using LLMs for software testing

Despite their potential, the use of LLMs in software testing involves specific and deeper challenges. One of the fundamental difficulties is the so-called oracle problem. In classical testing, a test oracle defines the expected, correct result for a given input. For generative models, however, the definition of such an oracle is problematic, as there is often no specific, correct answer. The outputs generated by LLMs are often creative, subjective, and diverse, which complicates unambiguous evaluation (Aleti, 2023).

Further challenges include:

- Correctness and Reliability: LLMs tend to produce so-called hallucinations, where they generate syntactically correct but semantically incorrect or nonsensical code. This can lead to faulty test cases, compilation errors, or incorrect assertions, which limits their reliability (Walczak et al., 2026).
- Probabilistic Nature: The results of LLMs are inherently probabilistic and not deterministic. The same query can lead to different results, which complicates the reproducibility and consistency of test generation (Jalil et al., 2023).
- Context Understanding: Although LLMs can process large contexts, they have difficulties fully capturing the entire project context, dependencies between modules, or specific domain knowledge, which can lead to incomplete or irrelevant tests (Yuan et al., 2024).

These challenges show that the simple application of LLMs is not sufficient. Rather, intelligent frameworks and methods are needed that leverage the strengths of the models while compensating for their weaknesses through validation, correction, and feedback mechanisms.

2.4 Test solutions and their evaluation

The application of generative AI and especially Large Language Models (LLMs) to software engineering tasks is increasingly being researched. We structured the description of the solutions from different perspectives (e.g. description of specific frameworks and tools or specific measures considered).

Empirical studies on the performance of LLMs A variety of studies have empirically investigated the ability of LLMs such as ChatGPT, Codex, and other GPT models to generate unit tests. A common finding of these studies is that LLMs can indeed produce human-like and well-readable tests, but also exhibit a high error rate. This includes compilation errors, the use of non-existent symbols, or faulty assertions that require manual post-processing (Yuan et al., 2024). Beyond executable unit test code, LLMs have also been evaluated for generating other types of testing artifacts such as test plans (Haldar & Capretz, 2026).

Specific frameworks and tools In response to the challenges of pure LLM application, specialized frameworks have been developed that embed LLMs into a structured process.

- TESTPILOT, presented by Schäfer et al. (2024), is an adaptive test generation system for JavaScript that feeds LLMs with carefully crafted prompts containing context information such as function signatures, documentation comments, and code examples. A central feature is an iterative repair mechanism, where failed tests are sent back to the LLM together with the error message to produce a corrected version.
- ChatTester, developed by Yuan et al. (2024), follows a similar approach and uses ChatGPT to generate tests to improve quality. The framework consists of an initial test generator that first captures the intention of the method to be tested through the LLM, and an iterative refiner that uses compilation errors to provide targeted feedback to the model for repair.
- ChatUniTest by Chen et al. (2024) also implements a cycle of generation, validation, and repair and uses an adaptive mechanism for generating context to optimize the relevance of the information provided to the LLM.

These frameworks show that integrating LLMs into an automated feedback cycle is a good way to increase the correctness of generated artifacts.

Various research groups have developed specialized tools to address specific problems in test generation:

- Meta ACH (Automated Compliance Hardening): Meta developed a mutation-driven system (Harman et al., 2025) that generates specific errors (mutants) and subsequently creates tests to ”kill” them. Engineers there accepted 73% of the tests generated by ACH, with 36% being classified as relevant for privacy aspects.
- UTFix for Test Repair: As software code constantly evolves, tests often become outdated. The tool UTFix (Rahman et al., 2025) uses LLMs to repair unit tests when the focal method changes (Change Aware Repair). On a synthetic benchmark, UTFix was able to fix 89.2% of assertion errors and restore 100% code coverage for 96 of 369 tests. In a real-world benchmark, 60% of assertion errors could be fixed and 100% code coverage restored for 19 of 30 unit tests.
- LspAi for Multi-Language Support: To standardize static analysis across different languages, the tool LspAi (Go et al., 2025) uses the Language Server Protocol (LSP). Compared to GitHub Copilot, LspAi achieved significant improvements in line coverage: +145% in Java, +931% in Golang, and +95.62% in Python.

Comparative studies with traditional methods Several studies have directly compared tests generated by LLMs with those from traditional Search-Based Software Testing (SBST) tools such as EvoSuite (Tang et al., 2024; Wang et al., 2024b; Yang et al., 2024) or Pynguin (Bhatia et al., 2024). The results show a mixed picture:

- In terms of code coverage, SBST tools often achieve better results, as they are explicitly optimized to maximize this metric (Yang et al., 2024).
- LLMs, however, significantly surpass traditional tools in the readability and understandability of generated tests. The tests are more similar to those written by human developers, which facilitates their maintenance and use in practice (Schäfer et al., 2024).

Hybrid approaches are also interesting, where the strengths of both worlds are combined, for example by using LLMs to generate starting populations for SBST algorithms. Bhatia et al. (2024) mentioned that a combination of Pynguin and ChatGPT could improve the creation of unit tests.

Correctness and compilability of generated tests One of the major hurdles for practical use of LLMs in test automation is the correctness of generated test cases (Yuan et al., 2024). The literature shows a very mixed picture here, which strongly depends on the LLM used, the complexity of the code to be tested, and the applied prompting strategy (Wang et al., 2024a). Correctness can be divided into several levels: syntactic correctness, compilability, and successful execution.

Studies examining direct test generation with ”out-of-the-box” approaches often report low success rates. A comprehensive study with ChatGPT for generating 1,000 Java unit tests found, for example, that only 42.1% of generated tests could be successfully compiled and only 24.8% passed subsequent execution (Yuan et al., 2024). Other investigations came to more positive results: One study reports that 69.6% of test cases generated by ChatGPT could be compiled and executed without human intervention (Tang et al., 2024). In an industrial case study at Meta, the generated test cases achieved a compilation rate of 75% (Alshahwan et al., 2024). AGONETEST achieved an average compilation rate of 75% with various GPT models (Lops et al., 2025). These figures show a discrepancy: While compilability is often relatively high, the rate of successfully executed tests (”Green Suite”) is usually significantly lower. In the Meta study, only 57% of tests passed reliably, and with AGONETEST it was only about 34%.

The analysis of faulty tests shows frequent error categories in the study by Yuan et al. (2024):

- Compilation errors: 57.9% of tests generated by ChatGPT had compilation errors. By far the most common cause there were ”Symbol Resolution Errors” (e.g., ”Cannot find symbol class/method/variable”). This indicates that LLMs often have difficulty correctly capturing the complete project context, dependencies, and visibility rules. Other common errors are type errors (”Incompatible types”) and access errors to private or protected members (”Private access”).
- Runtime errors and false assertions: A large proportion of compilable tests fail during execution. The vast majority of these errors (85.5% in one study) are caused by false

assertions. Here, the LLM misinterprets the intention of the method being tested and ”hallucinates” an expected behavior that does not correspond to reality. Another source of errors are runtime environment errors, such as ”NullPointerExceptions”, which occur when the LLM tries to access external resources (e.g., files) that do not exist in the test environment.

To address these correctness problems, modern approaches rely on iterative refinement and repair mechanisms. Systems such as the previously mentioned TESTPILOT (Schäfer et al., 2024), ChatTester (Yuan et al., 2024), and ChatUniTest (Chen et al., 2024) use a repair loop where errors from compilation or execution are analyzed and fed back to the LLM to generate a corrected version. This approach can increase the success rate: ChatTester was able to increase the rate of compilable tests by 34.3% and the rate of successfully executed tests by 18.7%. TestPilot was able to automatically repair an average of 15.6% of faulty tests.

Code coverage and bug detection In addition to pure correctness, code coverage and the ability to detect errors are important quality metrics for test suites. Research shows that tests generated by LLMs, especially those that are successfully executable, can achieve high code coverage, which is often comparable to or even superior to that of traditional tools or even manually written tests (Yuan et al., 2024).

The results on code coverage are inconsistent in the literature and strongly context-dependent. Some LLM systems show better performance compared to traditional approaches. The TESTPILOT system, for example, achieved a median statement coverage of 70.2% and a branch coverage of 52.8% for JavaScript packages, significantly outperforming the traditional tool Nessie (51.3% and 25.6%, respectively) (Schäfer et al., 2024). ChatUniTest also achieved higher overall line coverage at 59.6% than EvoSuite at 38.2% (Chen et al., 2024). Other studies come to opposite results. A comparative study for Java found that EvoSuite achieved higher average coverage than ChatGPT (74.2% vs. 55.4% statement coverage) (Tang et al., 2024). An important nuance is that successfully executed LLM tests often have very high coverage. One study found that these tests achieved comparable and sometimes even higher coverage than manual tests and EvoSuite (e.g., 82.3% statement and 65.6% branch coverage for ChatGPT vs. 84.2% and 68.9% for manual tests) (Yuan et al., 2024).

The ability to find real bugs is the actual goal of testing. LLM-generated tests have shown potential here, but are often still inferior to specialized tools. In a direct comparison on the Defects4J benchmark, one study found that ChatGPT found 44 of 212 bugs, while EvoSuite found 55 (Tang et al., 2024). This shows that LLMs are in principle able to uncover real errors, but effectiveness varies. One study found that the error detection capability of LLMs is limited, mainly due to the high error rate of the tests themselves. Of the errors for which valid tests were generated, only 47.28% were actually detected (Yang et al., 2024). A core problem remains the unreliability of generated assertions. LLMs tend to ”hallucinate” incorrect expected values, leading to a high rate of failing tests that do not uncover real bugs but are themselves faulty. This complicates the distinction between a real software error and an error in the test itself.

In summary, it can be stated that LLMs have the potential to generate tests with high coverage. However, their practical effectiveness in bug detection is currently still limited by the high rate of faulty generations, especially in the assertions.

Emerging approaches to Test Generation with LLMs The trend is moving away from single-pass approaches toward iterative multi-agent systems and hybrid methods:

- RAG and Context Enrichment: Simply passing method signatures is often insufficient. Recent approaches, such as the integration of Prompt Engineering and External Dependency Retrieval (EDR), where source code from external libraries is injected into the prompt, lead to significant improvements. One study showed that the combination of Prompt Engineering and EDR can increase coverage in industrial projects by an average of 12% (Lenke et al., 2026).
- Iterative Refinement: Frameworks like Retry and Refine (Ghosh & Abbas, 2026) utilize multi-agent architectures (based on LangGraph) to generate, validate, and regenerate tests based on feedback. For this purpose, they employ seven agents to handle various tasks such as user stories, validation, etc. Such pipelines were able to increase Pass@3 success rates on benchmarks like MBPP and HumanEval by up to 50 percentage points compared to baseline pipelines.

Focus on prompt engineering and context provision Research has quickly recognized that the quality of LLM results strongly depends on the quality of the input prompt, an area known as prompt engineering. Studies investigate which contextual information is most useful. This includes not only the source code of the method to be tested, but also class declarations, dependencies, API documentation, existing tests, and even error messages from failed compilation or execution attempts. Yang et al. (2024) conducted a comprehensive study on the influence of different prompt designs and context factors on the performance of LLMs and showed that careful tuning of the prompt can significantly increase effectiveness. It has, for example, been shown that more complex prompts (e.g., Chain-of-Thought) do not always deliver better results than baseline or rich-instruction prompts (Andrzejewski et al., 2025). Nevertheless, Guzu et al. (2025) states that providing context (problem description plus solution code) significantly increases the results.

Solutions for removal and repairs A separate but related challenge in automated test generation concerns the execution and handling of generated tests, particularly when tests contain errors or cause runtime failures. Different approaches have been developed to address this problem:

ChatUniTest (Chen et al., 2024) employs multiple repair strategies. Rule-based repairs handle specific errors such as missing closing symbols or missing imports (by copying import statements from the focal class). When these fail, LLM-based repair will be used to fix the error. However, this approach can be computationally expensive, often requiring multiple LLM iterations per test, and it is possible that some of tests remain unfixable even after multiple repair attempts.

Some studies address failing tests through removal rather than repair. Meta's study (Alshahwan et al., 2024) reports discarding tests that do not build immediately, focusing only on reliably passing tests. This approach could lose valuable test cases that could detect real bugs.

The Nessie system (Arteca et al., 2022) implements a try/catch wrapper pattern that would prevent individual test failures from terminating the entire test run. However, Nessie

has been criticized for generating tests not containing any assertions and counting coverage across all generated tests regardless of pass/fail status (Schäfer et al., 2024).

The trade-off between these approaches represents a fundamental design decision: repair-focused methods optimize for code quality but can incur high computational costs and cannot guarantee success, while removal strategies ensure pipeline stability at the cost of potentially valuable tests. Defensive wrapper approaches offer a middle ground but require careful consideration of assertion quality to maintain test validity.

Research gap for specific programming languages The analysis of related work shows a clear focus of research on a limited selection of widely used programming languages. The majority of empirical studies and developed frameworks focus primarily on Java (Yang et al., 2024), JavaScript (Schäfer et al., 2024) and Python (Bhatia et al., 2024). Studies have also been conducted for languages such as Kotlin (Alshahwan et al., 2024), C# and C++ (Paduraru et al., 2024), which expands the range but still remains limited to popular, often object-oriented or system-level languages. A gap in the current research landscape exists, however, for the Perl programming language. Despite reviewing a multitude of sources, no study could be identified that explicitly deals with the automated generation of unit tests for Perl using large language models. Perl systems are thus a good representation of legacy systems relying on languages not spread as widely, which likely has an impact on the training data used by LLMs. Recent Perl-specific LLM work has begun to address this imbalance from a training-data perspective. PerlGPT's MetaCPAN Curator is described as selecting high-quality CPAN distributions to build Perl-related stimulus/response pairs for fine-tuning a Code Llama or comparable model (Braswell, 2024). This shows growing interest in Perl-specific LLM adaptation, while the testing-focused use of LLMs for Perl remains underexplored.

Taken together, the reviewed literature directly motivates the design and evaluation focus of this work. The limitations of traditional SBST approaches motivate considering LLM-generated tests not only in terms of coverage, but also in terms of readability and practical maintainability (Schäfer et al., 2024; Tang et al., 2024). The frameworks TESTPILOT, ChatTester, and ChatUniTest motivate an automated feedback cycle in which tests are generated, executed, diagnosed, and repaired instead of relying on a single generation step (Chen et al., 2024; Schäfer et al., 2024; Yuan et al., 2024). The studies on prompt engineering and context provision motivate the use of structured context and error information as inputs for generation and repair prompts (Guzu et al., 2025; Yang et al., 2024). ChatTester's initial generator, which captures the intention of the method, motivates the business-requirements phase used in this work. Finally, the comparison of repair, removal, and defensive execution strategies motivates treating failed generated tests as a central design and evaluation issue rather than as an implementation detail (Alshahwan et al., 2024; Arteca et al., 2022; Chen et al., 2024). Since no prior work was identified for LLM-based unit-test generation in Perl, we apply these insights to a language setting that remains underrepresented in the existing literature.

3 Solution description

As highlighted in Section 1 the goal of the developed system is the automated generation of good unit tests for Perl code using modern Large Language Models (LLMs). The challenge is to generate not only syntactically correct but also semantically meaningful tests that cover the actual business logic and error conditions of legacy code.

The architecture was designed modularly to reduce the complexity of individual subprocesses and enable clear separation of responsibilities. An approach where the LLM is directly confronted with the entire code and a simple test instruction leads to high error rates and unreliable results in practice. The sources and own experiments show that LLMs tend toward hallucinations, loss of context and semantic errors in complex tasks like test generation when context is not precisely and structuredly specified (Chen et al., 2024; Yuan et al., 2024). For this reason, the system relies on multi-stage processing: First, the code is analyzed and context extracted, then targeted prompts for business logic and test generation are created before results are validated and repaired if necessary. This architecture reduces error sources and enables iterative quality improvement.

3.1 Overview of the workflow and technology stack

The overall system consists of several distinct main components that work together in this workflow. Figure 1 shows the complete workflow of the developed system from code analysis to final quality assessment.

1. **Analyze the test object ❶:** The process begins at the top left with two input modes: Either the entire repository is analyzed (mode *Generation of Tests*), or only changed functions are specifically identified (mode *git diff*). The resulting Perl code to be tested is analyzed with a static parser. All relevant function definitions, dependencies, and particularities are extracted. This information forms the basis for subsequent steps.
2. **Unit test generation ❷:** The extracted context data is used to create structured prompts for the LLM. The prompts are designed to address the business logic and technical requirements of the target function. Communication with the Large Language Model occurs via a Python component that sends prompts to the model via API and processes responses.
3. **Test execution ❸:** The generated tests are automatically executed and results analyzed.
4. **Correct faulty tests ❹:** Errors are classified and diagnosed to enable targeted repair measures. Failed tests are repaired by incorporating error diagnostics into new prompts and requesting correction from the LLM. The exact number of repetitions can be adjusted by developers as needed.
5. **Test analysis ❺:** Documentation regarding test failures is created and code coverage is measured.

The data flow between components is designed so that each step builds on the results of the previous one. The modular structure simplifies extensibility and adaptation of the system to new requirements or other programming languages. The entire architecture is designed to detect and fix errors early and continuously improve the quality of generated tests.

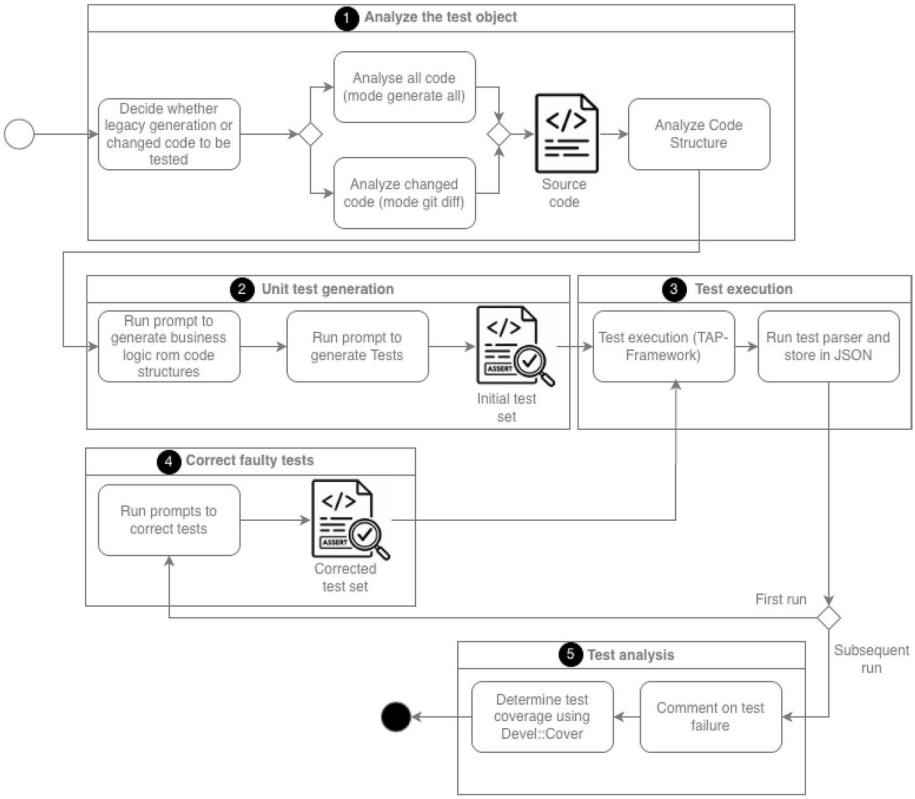

Fig. 1 Automated test generation workflow

Documentation occurs with each run, ensuring traceability of all steps and results for the application (see ❸ in Fig. 1). The execution level includes detailed logging of all LLM interactions and system operations. This documentation captures both the prompts sent and the responses received with timestamps. At the evaluation level, structured JSON logging of all test results occurs. This machine-readable documentation contains all quantitative metrics and enables statistical analyses.

The developed system follows a modular architecture with defined interfaces between components. Implementation was carried out in Python for generation and analysis, and in Perl for language-specific analysis tools.

The system is modularly designed and uses the following technologies and interfaces. The main logic and analysis are implemented in Python 3.12, while Perl 5.32.1 is used for static analysis and Perl 5.26.3 for test execution and coverage measurement. Containerized execution with Podman-Compose ensures a reproducible environment and prevents problems from local installations. IBM WatsonX (IBM, 2025) is used as the LLM provider. Communication occurs via structured JSON files and API calls. For test execution, the `Test2::V0` (Granum, 2025) framework 0.000139 is used, for code coverage `Devel::Cover` (Johnson, 2025) 1.36. The architecture is designed so that all components are clearly separated and extensible.

Communication with LLMs occurs through IBM WatsonX. The API supports asynchronous and parallel requests, so generation for multiple functions can occur simultaneously. Errors such as timeouts or rate limits are caught by a retry and backoff system. Hyperparameters were configured informed by prior empirical findings on the impact of hyperparameters on code generation performance (Arora et al., 2024). Specifically, `temperature` was set to 0.3 and `top_p` to 0.2 to reduce stochasticity and bias the model toward high-probability continuations. The `max_tokens` were set to 10,000 based on experience with the inputs. The LLM responses are automatically processed further and transferred into test files.

In the following sections we describe each step in detail.

3.2 Analyze the test object

The automated test generation begins with a complete static analysis of the Perl code (see ❶ in Fig. 1). The script processes all relevant module files and identifies the existing function definitions. For each function, the signature, the parameters used, and the internal control structures are extracted. Furthermore, all dependencies, such as `use` and `require` statements, as well as global variables and special blocks like `BEGIN` are recognized. The analysis delivers not only a list of functions, but also complexity metrics, such as the number of branches and maximum nesting depth. References to constructs such as `eval`, file operations, or exception handling are also documented. The results are stored as a structured JSON block and form the basis for the subsequent steps.

3.3 Unit test generation

The extracted context information is processed in a multi-stage prompt process to derive the unit test cases (see ❷ in Fig. 1). We divide unit test generation into two phases.

In the first phase, the LLM receives a prompt that should analyze the business logic and the expected input and output behavior of the target function, the main code is also sent here. The prompts are designed to point out Perl-specific peculiarities such as automatic type conversion and context-dependent operators.

In the second phase, the LLM is instructed to generate Test2::V0 (Granum, 2025) compliant unit tests based on the previously extracted business logic. The prompts contain clear format specifications, such as the requirement for a final done_testing(); and explicit checks for the existence of symbols to avoid errors such as multiple mocking. The prompts are assembled from the extracted context data and fixed template files. Error diagnostics from previous test runs can be used as context for repair prompts (see Step ❹), whereby an intermediate step also processes another prompt that draws attention to the errors in a structured way.

The creation of prompts for unit test generation follows a structured approach to ensure robust tests. The template defines rules that ensure all code paths, edge cases, and dependencies are covered.

3.3.1 Phase 1: Run prompts to generate business logic from code structures

As the first phase in the multi-stage prompt process, a dedicated prompt is used to analyze the business logic of the function to be tested. This prompt receives both the complete module context and the specific function and extracts the following information:

- **Purpose**: The role of the module and the specific function in the overall system
- **Inputs/Outputs**: Expected parameters, return values, and data types
- **Edge Cases**: Boundary cases and special conditions that need to be tested
- **Error Handling**: How the function handles invalid inputs or error situations
- **Architecture Integration**: How the function fits in the overall module structure
- **Dependencies**: The Dependencies or interactions with other modules in the function

The output of this prompt is incorporated as structured context into the subsequent test generation prompt. This is intended to help the LLM generate tests that are not only technically correct but also cover the actual business logic and intended functionality. The Business Logic Prompt thus forms the bridge between the often implicit domain knowledge in the code and the explicit test assertions. That is, in order to write test assertions we need the desired behavior (test oracle). Given that legacy requirements are missing, thus a dedicated step generates business logic, including requirements.

3.3.2 Phase 2: Run prompt to generate test cases

In the second phase, we construct the concrete test cases. The template provides rules regarding different aspects of test generation (e.g. Perl-specific constructs, test structure and outputs). These aspects are described in the following.

Perl constructs To avoid compilation errors, the prompt template for test generation contains explicit rules for critical Perl constructs:

- **Module Loading**: Modules must be loaded with `BEGIN { require ModuleName; }`, never with file paths like `require 'Path/Module.pm'`.
- **Variable Declaration**: All variables must be declared with `my` (`my $result = eval { ... };`) to ensure `use strict` compliance.
- **String Safety**: Strings with special characters like `@`, `$`, or `%` must be in single quotes (`my $email = 'test@example.com';`) to prevent unwanted interpolation.
- **Function Calls**: System functions like `rmdir` or `unlink` must not be called with a preceding `$`.
- **Constants**: Constants must be fully qualified (`Module::CONSTANT`) to avoid "Bareword not allowed" errors.

Test structure The template defines a standardized test structure for all generated tests, as outlined in Listing 1.

Listing 1 Universal Test Structure

```
use Test2::V0 -no_srand => 1;
use strict;
use warnings;
use Test2::Mock qw/mock/;
use File::Temp qw/tempfile tempdir/;
use File::Spec;
use lib '.';
BEGIN { require ModuleName; }  # Use actual module name from package declaration

# Function existence check
my $symbol_check = eval { no strict 'refs'; defined &{"ModuleName::function_name"} };
if ($@) { fail('Symbol check crashed: ' . $@); } else { ok($symbol_check, 'function_name is defined'); }

# Test cases with eval protection
my $result = eval { ModuleName::function_name(@args) };
if ($@) { fail('Function crashed: ' . $@); } else { ok(defined $result, 'Function returns result'); }

# More test cases...

done_testing();
```

This structure ensures that tests can continue running even if individual function calls fail.

Test strategy The template defines the following test principles:

- **Coverage Requirements**: All code branches, edge cases, error conditions, invalid outputs and normal operations must be tested.
- **Error Handling**: All function calls must be enclosed in `eval` blocks so that an error does not prevent all subsequent tests. Error messages should be output with `fail()` and descriptive texts.
- **Mocking Strategy**: External dependencies (databases, HTTP, files) must be mocked. Complex object methods are mocked when needed, but never the test framework itself.
- **File Operations**: Temporary files and directories are created with `File::Temp` and properly cleaned up.
- **Assertions**: Clear, descriptive test names are mandatory. `is()` is preferred over `ok()` when checking specific values. Comments explain the test purpose.

Output specification The template specifies the following requirements for generated tests:

- A single .t file per function
- Must compile without errors under `use strict`
- No subtests, no `plan()`, only `done_testing()`
- Completely self-contained and executable
- Helpful comments for documentation

3.4 Test execution

The generated test cases are saved according to a clear file naming convention that enables unambiguous assignment to module and function. Test execution occurs in a container, which ensures parallel test runs and a consistent environment. The outputs are analyzed by a Test Anything Protocol (TAP) parser (see ❸ in Fig. 1) that extracts individual test assertions and their status, structurally assigns diagnostics and error messages, and distinguishes between compilation errors, runtime errors, and faulty assertions. Test2::V0-specific comparison tables and exception reports are also read out. The results are saved as JSON and serve as the basis for the repair phase.

3.5 Correct faulty tests

For the correction of faulty tests (see ❹ in Fig. 1), we construct a repair prompt template and an error analysis template.

3.5.1 Prompts

Repair template For automatic repair of faulty tests, a separate prompt template exists that combines the following elements:

- **Business Requirements**: The previously extracted business logic as context
- **Error Analysis**: A structured analysis of errors from a preceding analysis prompt
- **Source Code**: The original Perl code of the function to be tested
- **Failed Test Code**: The failed test code
- **Test Output**: The complete output with diagnostics
- **Working Examples**: Optional successful tests as reference

The repair template emphasizes critical aspects such as the correct use of `eval` blocks, mocking dependencies with conditional checking for symbol existence. Particularly important is the instruction to use manual `eval` blocks instead of `dies{}`/`lives{}` helpers, as these can interfere with test isolation.

Error analysis template An additional analysis prompt is used to structure complex error messages and prepare them for the repair prompt. This prompt identifies error types (compilation, runtime, assertion, missing dependencies), extracts relevant error messages, summarizes probable causes, and suggests concrete solution hints.

This prompt structure is designed so that the generated tests are not only syntactically correct but also semantically meaningful, checking both the functionality and stability of the code.

3.5.2 Defensive eval wrapper strategy

A distinctive methodological feature of the developed approach is the systematic use of Perl's `eval` block to wrap all function calls to the code under test. This design decision addresses a critical challenge in automated test generation: preventing individual test failures from terminating the entire test execution.

Rationale LLM-generated tests frequently contain calls to functions that may crash, throw unexpected exceptions, or behave unpredictably due to hallucinated assumptions about the code's behavior. Without defensive wrapping, a single runtime error can abort the test file, losing results from the assertions that follow it and leaving the test suite incomplete in automated pipelines.

The eval wrapper pattern ensures that:

- A crash in the tested function is captured as a controlled failure (`fail()`) rather than terminating the test process
- Subsequent assertions and teardown operations continue executing
- Execution continues past the crash, so the remaining assertions in the file still produce results
- Error diagnostics are preserved for analysis and repair

The wrapping applies to calls to the code under test. Setup code such as mock configuration is not wrapped, so a fault there can still terminate a file before any assertion is reported.

Comparison with alternative approaches This strategy differs fundamentally from approaches in the literature:

- **Repair-focused approaches** These attempt to fix failing tests through rule based approaches and iterative LLM calls, these can increase computational cost. Our wrapper makes tests immediately executable without additional LLM inference.
- **Removal-based approaches** (Meta's industrial study) discarding tests that do not build immediately. This loses potentially valuable tests that could detect real bugs through their failure behavior. Our approach retains these tests and records their failures.
- **Nessie's approach** does not employ LLMs and uses similar try/catch isolation but has been criticized for generating tests without meaningful assertions and counting coverage regardless of test outcomes.

Our generated tests contain explicit assertions that verify expected behavior. The eval wrapper serves solely as a safety mechanism that captures unexpected runtime exceptions rather than serving as a means to inflate coverage metrics or mask test failures. Specifically, the wrapper intercepts exceptions raised during function execution (e.g., undefined subroutine calls or explicit die statements). When the function executes successfully and returns a value, control proceeds to the else branch where assertions are evaluated. Assertion outcomes are handled by the Test2::V0 framework and reported as standard test failures. This

multi-stage structure ensures that the eval wrapper provides crash resilience while preserving the semantic outcomes of assertions.

Implications for coverage metrics The use of defensive wrappers has implications for interpreting coverage metrics. Devel::Cover measures executed statements, not assertion success. Through the wrapper, more statements may be executed even when tests fail, potentially yielding higher coverage values than approaches where failing tests abort execution. The system addresses this concern through a dedicated operating mode that comments out failing tests before coverage measurement, ensuring that only passing tests contribute to coverage metrics (see Section 3.5.3).

3.5.3 Modes for test generation and error handling

The system also offers two operating modes that address the needs of developers:

- **comment-failures:** In this mode, all failing tests are automatically commented out after execution. This is particularly helpful when developers want to generate a basic test foundation without immediately having to correct all errors. The commented-out tests remain visible in the code and can be activated and improved later.
- **git diff:** This mode uses the results of a `git diff` analysis to specifically identify new, changed, or deleted functions. New tests are generated for these functions. This ensures that the test base always remains current and responds to code changes.

These modes improve the flexibility of the system and support different development strategies, from rapid test base creation to quality assurance for code changes.

3.6 Test analysis

The test analysis phase (see ❺ in Fig. 1) performs two activities.

Commenting on test failures All tests that still fail after repair are automatically commented out in the test files. This corresponds to the *comment-failures* mode described in Section 3.5.3. Commenting works at the level of individual assertions rather than entire test files. Everything not observed passing during that run is commented out, including failing assertions, assertions that were never executed, and any code following the last passing assertion. This ensures that assertions previously blocked by a crash do not become active and inflate the measurement. The commented-out tests remain visible in the source code so that developers can inspect, manually fix, and reactivate them later, while the remaining passing tests form a stable, executable test suite.

Determining test coverage After the failing tests have been commented out, code coverage is measured using `Devel::Cover`. Because only passing tests are active at this point, the reported statement, branch, condition, and subroutine coverage values reflect genuinely exercised and verified code paths. This prevents potential inflation of coverage metrics through code that is merely executed but whose assertions fail.

3.7 Example: Complete test generation process

To illustrate the entire process, a simple Calculator module is used as an example. This example shows how the system works from code analysis through prompt construction to final test generation. The prompt templates used are documented in Online Resource 1, the complete generation log with all details on the process flow can be found in Online Resource 2.

3.7.1 Input module

The following Perl module implements two simple mathematical operations:

Listing 2 Calculator.pm - Example Module for Test Generation

```
package Calculator;

use strict;
use warnings;


sub add {
    my ($a, $b) = @_;
    return $a + $b;
}

sub subtract {
    my ($a, $b) = @_;
    return $a - $b;
}

1;
```

3.7.2 Generated test

The system analyzes this module, extracts the two functions `add` and `subtract`, and automatically generates corresponding unit tests. The following code shows the complete generated test for the `add` function:

Listing 3 test_Calculator_add.t - Test Generated by the LLM

```
use Test2::V0 -no_srand => 1;
use strict;
use warnings;
use Test2::Mock qw/mock/;
use File::Temp qw/tempfile tempdir/;
use File::Spec;
use lib '.';
BEGIN { require Calculator; }

# Function existence check
my $symbol_check = eval { no strict 'refs'; defined &{"
    Calculator::add"} };
if ($@) { fail('Symbol check crashed: ' . $@); } else {
    ok($symbol_check, 'add is defined'); }

# Test case: Normal addition
my $result = eval { Calculator::add(2, 3) };
if ($@) { fail('Function crashed: ' . $@); } else { is(
    $result, 5, 'Normal addition works'); }

# Test case: Addition with negative numbers
$result = eval { Calculator::add(-2, 3) };
if ($@) { fail('Function crashed: ' . $@); } else { is(
    $result, 1, 'Addition with negative numbers works');
    }

# Test case: Addition with decimal numbers
$result = eval { Calculator::add(2.5, 3.7) };
if ($@) { fail('Function crashed: ' . $@); } else { is(
    $result, 6.2, 'Addition with decimal numbers works');
     }

# Test case: Addition with zero
$result = eval { Calculator::add(0, 5) };
if ($@) { fail('Function crashed: ' . $@); } else { is(
    $result, 5, 'Addition with zero works'); }

# Test case: Non-numeric input
$result = eval { Calculator::add('a', 5) };
if ($@) { fail('Function crashed: ' . $@); } else { isnt
    ($result, 5, 'Non-numeric input does not work'); }

done_testing();
```

3.7.3 Analysis of the generated test

The generated test demonstrates several best practices of test development:

1. **Function Existence Check**: Before tests are executed, it is checked whether the function is defined at all. This prevents unclear error messages for missing functions.
2. **Error Handling**: All test cases are enclosed in `eval` blocks to catch runtime errors and generate meaningful error messages. Importantly, the eval wrapper only catches *runtime exceptions* during function execution. When the function executes successfully, control passes to the `else` branch where assertions are evaluated by Test2::V0. This multi-stage

architecture is demonstrated by Test 6: `Calculator::add('a', 5)` executes successfully without any exception (the eval block completes normally), then the assertion `isnt($result, 5)` is evaluated and fails because Perl's implicit string-to-number conversion yields `0 + 5 = 5`. This assertion failure is properly reported by Test2::V0 as a failed test, not masked by the eval wrapper. The wrapper thus provides crash resilience without suppressing assertion outcomes.

3. **Comprehensive Test Cases**: The LLM has automatically identified various scenarios:
   - Normal cases (positive integers)
   - Edge cases (negative numbers, zero, decimal numbers)
   - Error cases (non-numeric inputs)
4. **Clear Test Descriptions**: Each test has a meaningful description that explains the tested behavior.
5. **Test2::V0 Idioms**: The test uses Test2::V0 assertions (`ok`, `is`, `isnt`) and follows framework conventions.

This example shows that the LLM is able to generate not only syntactically correct tests, but also identify semantically meaningful test cases that cover various aspects of functionality.

3.7.4 Test results and execution statistics

After generation, the tests were automatically executed and analyzed. The system provides detailed JSON output with structured information about each individual test.

For the `add` function, 6 tests were generated, of which 5 were successful (Table 1):

The failed test (No. 6) illustrates an interesting characteristic of Perl: The function does not throw an exception for non-numeric inputs, but implicitly converts the string `'a'` to `0`. The test expects that the result is *not* equal to 5 (`isnt($result, 5)`), however `0 + 5 = 5`, causing the assertion to fail.

The error diagnostics show:

```
Failed test 'Non-numeric input does not work'
# at t/generated/test_Calculator_add.t line 32.
Comparison matched (it should not).
```

This example shows both a strength and a limitation of LLM-based test generation:

- **Strength**: The LLM automatically recognizes that non-numeric inputs are an important test case and generates corresponding tests.
- **Limitation**: The LLM has not fully considered Perl's implicit type conversion and for-

Table 1 Test results for test_Calculator_add.t

Nr.	Test Description	Status
1	add is defined	OK
2	Normal addition works	OK
3	Addition with negative numbers works	OK
4	Addition with decimal numbers works	OK
5	Addition with zero works	OK
6	Non-numeric input does not work	FAIL

mulated an assertion based on incorrect assumptions about runtime behavior. But it could also be an unwanted behavior for a calculator which the developer might want to fix in the code.

The overall statistics for both generated tests show:

- **Total Tests**: 13 (6 for `add`, 7 for `subtract`)
- **Passed**: 11 (84.6%)
- **Failed**: 2 (15.4%)

These results show a high initial success rate of 84.6% for a fully automatic approach without manual post-processing. The detailed error diagnostics and the complete test generation process with all log entries are documented in Online Resource 2.

4 Evaluation

We first describe the research method, followed by the results of the evaluation.

4.1 Research method

The evaluation design follows an approach in which two different Large Language Models (operated locally due to digital sovereignty) are applied to the same Perl repositories. To mitigate the threat of randomness in LLM outputs, we performed each run three times, following the approach used in prior work (Yuan et al., 2024). First, test generation, then execution with automatic repair of failed tests, and finally analysis using the coverage metric. The same configuration is used for the industrial repositories and for the CPAN distributions.

4.1.1 Objectives and research questions

The evaluation addresses the research questions formulated in Section 1 as follows:

- **RQ1 (related to O1):***To what extent can LLM-based approaches automatically generate unit tests for legacy Perl modules that are syntactically correct and can be executed in a real test environment?* This is evaluated by analyzing the pass rates (proportion of successfully executed tests), the `no_output` error rate (tests that cannot be executed) as well as the number of generated tests. The evaluation includes both initial generation and the results after automatic repair.
- **RQ2 (related to O2):***How effective are repair mechanisms for solving problems?* This is investigated by comparing the error distribution before and after automatic repair. In particular, it is analyzed how effectively structural problems (`no_output` errors) and test failures (`test_failure`) can be addressed through repair.
- **RQ3 (related to O3):***How well do the generated and repaired unit tests cover the code of the systems under test?* In the analysis we focus on different typical coverage metrics (Statement, Branch, Condition, Subroutine).

The questions above are answered comparing Llama-3.3-70B-Instruct (70B parameters) and Mistral-Small-3.1-24B-Instruct-2503 (24B parameters).

4.1.2 Subjects - industrial systems

As evaluation basis, four productive Perl code repositories with a total of 186 functions from IBM were selected that show various aspects of legacy systems:

- **ACL (Access Control List)**: A module for managing access rights with 65 functions, distributed across two main modules (`Utils::ACL` and `Utils::ACL::Entry`). This repository presents an interesting test task with its rights logic.
- **Common (Utilities)**: A collection of 51 utility functions in the `Utils::Common` module that provide basic routine operations. The variety of functionalities makes this repository a good test for the generalization capability of the system.
- **DB2**: A database interface with 43 functions, distributed across four modules (`Utils::DB2`, `Utils::DB2::DiffTables`, `Utils::DB2::Measurements`, `Utils::DB2::Requests`). The database interactions and complex data structures pose requirements on test generation.
- **Log**: A logging framework with 27 functions in the `Utils::Log` module. This smaller repository tests how well the LLMs are familiar with logging functions.

All modules come from productive systems and contain legacy code characteristics such as grown structures, implicit dependencies, and missing tests. The source of these repositories is proprietary and cannot be released. A second set of subjects covers publicly available Perl systems.

4.1.3 Subjects - open-source systems

In addition to the industrial repositories, three publicly available CPAN distributions with a total of 139 functions were selected. The aim is to check whether the workflow also works outside a single organization's codebase and development environment. All three distributions are between 23 and 29 years old and have not received a new release in 8 to 13 years. MetaCPAN (The MetaCPAN Project, 2026) currently reports a bus factor of zero for each of them, meaning that no maintainer with indexing permission has published a release in the last two years. At the same time, they remain widely used: hundreds to thousands of other distributions depend on each of them, directly or indirectly. Because the source is public, the prompts, generated tests, and results are published in full.

- **XML::Simple 2.25** (McLean, 2026): A simple API layer over an underlying XML parser (`XML::Parser` or a SAX2 module), with 43 functions in the `XML::Simple` module and 786 dependent distributions. The documentation itself states that use in new code is "strongly discouraged" and that "new features are unlikely to be added". It also names the large number of options and the arbitrary ways in which they interact as the module's major problems, which gives this distribution the largest configuration surface of the three.
- **Tie::IxHash 1.23** (Sarathy & Ciornii, 2026): An implementation of ordered associative

arrays that preserve insertion order, with 28 functions in the `Tie::IxHash` module and 2,265 dependent distributions. The changelog dates the first release only as "ancient". The module is realized through Perl's tie interface (`TIEHASH`, `FETCH`, `STORE` and related methods) and therefore tests how well the models handle an indirect, callback-driven API.

- **Path::Class 0.37** (Williams, 2026): A cross-platform abstraction for file and directory path manipulation with 68 functions, distributed across four modules (`Path::Class`, `Path::Class::Dir`, `Path::Class::File`, `Path::Class::Entity`), and 2,645 dependent distributions. `Dir` and `File` both inherit from `Entity`, and many operations act on the filesystem, which makes test isolation harder.

At 46.3 functions on average, the three distributions are comparable in scale to the four industrial repositories (46.5).

4.1.4 Comparison of two large language models

For the evaluation, two different LLM models were deployed via the IBM WatsonX platform:

1. **Llama-3.3-70B-Instruct** (Meta, 2025): A large language model with 70 billion parameters from Meta, published in December 2024.
2. **Mistral-Small-3.1-24B-Instruct-2503** (Mistral, 2025): A smaller model with 24 billion parameters from Mistral, published in March 2025.

The comparison of these two models enables statements about the generalizability of the developed approach and the dependency of result quality on model size and architecture.

4.1.5 Data collected

Logging takes place as part of the test analysis step in our solution. We present the measures captured and how they are analyzed.

Correctness (RQ1) The correctness of generated tests is measured at two levels:

- **Test Pass Rate (TPR)**: The proportion of passing tests among all tests recorded for a run. Test files that produce no output, or that exit abnormally after reporting, are recorded as failed tests and count against the pass rate rather than being excluded from it. This metric considers both syntactic and semantic correctness, as tests only pass if the assertions are correctly formulated and the tested functionality meets expectations.
- **Error Type Rate (ETR)**: The distribution of error types across three categories: `no_output` (test files that produce no executable output, indicating structural problems such as compilation failures), `test_failure` (tests that execute but fail one or more assertions), and `unknown` (failures not classifiable into either category)

Repair effectiveness (RQ2) To quantify the effectiveness of the automated repair mechanism in relation to the **TPR**, we define *Repair Effectiveness* (RE) as the ratio of additional passing tests gained through repair to the number of initially failing tests:

$$\mathrm{RE} = \frac{\mathrm{Passed}_{\mathrm{repair}} - \mathrm{Passed}_{\mathrm{initial}}}{\mathrm{Failed}_{\mathrm{initial}}} \tag{1}$$

An RE value of 0 indicates that repair produced no additional passing tests, and negative values indicate that fewer tests passed after repair than before, that is, a net regression introduced during the repair step. RE is a net outcome measure rather than a count of repaired tests, because the number of executed tests is not fixed across the two runs. It can grow in two ways: the model may write additional assertions during repair, and assertions that were already present may become reachable once an earlier failure in the same file no longer aborts execution. RE can therefore exceed 1.0, since the additional passing tests are counted against the initial failures only.

To assess how repair affects the composition of errors (**ETR**), we report the absolute change in count per error category between the initial generation and post-repair results:

$$\Delta_c = \mathrm{Count}_{c,\mathrm{repair}} - \mathrm{Count}_{c,\mathrm{initial}} \tag{2}$$

where $c \in \{\texttt{test_failure}, \texttt{no_output}, \texttt{unknown}\}$. A negative Δ_c indicates that repair reduced the number of errors in category c, whereas a positive value indicates an increase. Since the number of executed tests may grow during repair, an increase in `test_failure` count does not necessarily signal degradation—it may reflect tests that now execute and fail semantically rather than not executing at all. Conversely, a decrease in `no_output` errors specifically indicates that repair resolved structural defects such as compilation failures.

Code coverage (RQ3) In the article by Obaid Barraood et al. (2023), it was investigated which metrics are used to evaluate test case quality; it was shown that particularly coverage metrics (such as statement, code, and branch coverage) were most frequently employed. Code coverage is measured with the Perl tool `Devel::Cover` and includes the following dimensions:

- **Statement Coverage (stmt)**: Proportion of executed statements
- **Branch Coverage (bran)**: Proportion of traversed branches
- **Condition Coverage (cond)**: Proportion of evaluated conditions
- **Subroutine Coverage (sub)**: Proportion of called subroutines

4.1.6 Execution

For the study, the automated test generation workflow has been run through completely from analysis of the test object to correction of faulty tests and test analysis. We logged the results for the initial generation, and the results after having corrected the faults. To explore the stability across runs, we followed the approach by Yuan et al. (2024) and conducted three workflow iterations. Coverage is calculated after generation and repair is concluded.

4.1.7 Threats to validity

Internal validity The prompt template was developed iteratively with Llama-3.3-70B in response to observed issues with compilation and test coverage. This introduces a prompt-development bias that should be considered when interpreting the cross-model comparison: the prompt templates may be better aligned with Llama-3.3-70B's response style and capabilities than with those of Mistral-Small-3.1-24B. At the same time, the template encodes domain-specific Perl testing rules, framework requirements, and best practices, rather than merely model-specific linguistic patterns. Consequently, the comparison evaluates both models under the same prompt template, but it cannot fully disentangle inherent model capability from prompt-template fit. Another limitation concerns the comparison between generation and repair: only the repair template provides a conditional mocking pattern to protect the setup. The reduction in structural errors after repair therefore cannot be attributed to the error feedback alone. A further internal-validity limitation is that no prompt-pipeline ablation was performed. In particular, we did not include a single-phase prompt baseline without the business-logic extraction phase. Therefore, the results support the effectiveness of the complete workflow, but not a causal claim about the independent benefit of Phase 1.

External validity The evaluation is based on four Perl repositories from a single company (IBM) and three publicly available CPAN distributions. Although these subjects represent different domains and complexity levels, comparability to other codebases, development styles, or organizations is limited. In particular, coding standards, documentation quality, and architectural patterns at IBM may have influenced the industrial results. The CPAN distributions reduce this threat, since they were written outside IBM. The results are specific to Perl and the Test2::V0 framework. Transferability to other programming languages, test frameworks, or paradigms (e.g., object-oriented vs. functional) is not readily given. The particular characteristics of Perl (context-dependent operators, implicit types, flexible syntax) could both complicate and facilitate test generation compared to other languages.

Construct validity The use of coverage metrics as a quality measure has known limitations. High coverage does not guarantee that the right things are being tested or that assertions are meaningful. A test could execute code without actually containing relevant assertions and would still contribute to coverage. The categorization of errors into test_failure, no_output, and unknown is based on automatic analysis of test outputs. This classification is not always unambiguous. The use of defensive eval wrappers affects coverage measurement. Since Devel::Cover counts executed statements regardless of assertion outcomes, coverage values could theoretically be higher than in approaches where failing tests abort execution. However, this is addressed by commenting out failing tests before coverage measurement, ensuring that only passing tests contribute to coverage metrics. Direct comparison with studies using different execution strategies still requires caution. The defensive wrapper strategy prioritizes pipeline stability and test retention over pure code quality. While this is advantageous for Continuous Integration and Continuous Delivery/Deployment (CI/CD) pipelines, it differs from repair-focused approaches that optimize for clean, human-readable test code.

4.2 Results

For each repository, three independent runs with the same configuration were performed to evaluate the stability and variability of the results. The following sections show the results regarding the research questions.

4.2.1 Correctness (RQ1) - industrial systems

Test Pass Rate (TPR) Table 2 shows the results of initial test generation without repair for both models across all three runs and four repositories.

The results show interesting patterns in variability. For Llama-3.3-70B, the variability is clearly visible for the ACL repository: The number of generated tests varies between 319 and 340, pass rates between 79.6% and 82.1%. For Common, there is moderate variability in test count (192–214 tests) and pass rate (74.0%–76.2%). Interestingly, DB2 and Log also show variability, both in test count and pass rate.

For Mistral-Small-3.1-24B, the variability is similarly pronounced: ACL fluctuates between 117 and 130 tests (71.9%–77.7% pass rate), Common between 165 and 176 tests (67.0%–67.5%), DB2 between 121 and 130 tests (68.8%–76.2%), and Log between 60 and 63 tests (65.0%–66.7%).

Error Type Rate (ETR) An analysis of error types shows patterns in failed tests. Table 3 shows the distribution of error types for all three runs of initial generation.

The error distribution shows clear and consistent differences between the models. For Llama-3.3-70B, there are mostly regular `test_failure` errors (54%–100% of errors across all runs), which suggests that the tests are syntactically correct and can be executed, but the assertions are not met. The consistency between runs is high: For ACL, the error types remain nearly identical across all three runs (98%–100% test_failure), while Common shows moderate fluctuations (54%–61% test_failure, 33%–44% no_output).

For Mistral-Small-3.1-24B, however, the `no_output` error type occurs frequently and consistently (43%–71% of errors), which indicates test files that could not be executed at all or produced no TAP-compliant output. This points to general code generation problems. The error distribution for Mistral-Small-3.1-24B is very stable across runs: For Log, the percentages are identical across Run 2 and 3 (14% test_failure, 71% no_output), DB2 also shows very consistent values (45%–47% test_failure, 46%–52% no_output).

4.2.2 Repair (RQ2) - industrial systems

In this section we analyze the repair performance from the TPR and ETR perspectives.

Test Pass Rate (TPR) Table 4 shows the results of the passed and failed tests after repair, including percentages. In addition, we added a column calculating the repair effectiveness introduced earlier.

Table 2 Initial test generation on industrial repositories across three runs

Repository	Model	Run	Tests	Passed	Failed
ACL	Llama-3.3-70B	1	319	254 (79.6%)	65 (20.4%)
		2	319	262 (82.1%)	57 (17.9%)
		3	340	271 (79.7%)	69 (20.3%)
	Mistral-Small-3.1-24B	1	130	101 (77.7%)	29 (22.3%)
		2	117	90 (76.9%)	27 (23.1%)
		3	121	87 (71.9%)	34 (28.1%)
Common	Llama-3.3-70B	1	214	163 (76.2%)	51 (23.8%)
		2	192	142 (74.0%)	50 (26.0%)
		3	195	146 (74.9%)	49 (25.1%)
	Mistral-Small-3.1-24B	1	166	112 (67.5%)	54 (32.5%)
		2	176	118 (67.0%)	58 (33.0%)
		3	165	111 (67.3%)	54 (32.7%)
DB2	Llama-3.3-70B	1	199	132 (66.3%)	67 (33.7%)
		2	187	133 (71.1%)	54 (28.9%)
		3	182	130 (71.4%)	52 (28.6%)
	Mistral-Small-3.1-24B	1	125	86 (68.8%)	39 (31.2%)
		2	121	89 (73.6%)	32 (26.4%)
		3	130	99 (76.2%)	31 (23.8%)
Log	Llama-3.3-70B	1	123	94 (76.4%)	29 (23.6%)
		2	133	103 (77.4%)	30 (22.6%)
		3	130	107 (82.3%)	23 (17.7%)
	Mistral-Small-3.1-24B	1	60	40 (66.7%)	20 (33.3%)
		2	63	42 (66.7%)	21 (33.3%)
		3	60	39 (65.0%)	21 (35.0%)

Table 3 Distribution of error types in initial generation on industrial repositories (all runs)

Repo	Model	Run	test_failure	no_output	unknown
ACL	Llama-3.3-70B	1	65 (100.0%)	0 (0.0%)	0 (0.0%)
		2	56 (98.2%)	0 (0.0%)	1 (1.8%)
		3	69 (100.0%)	0 (0.0%)	0 (0.0%)
	Mistral-Small-3.1-24B	1	8 (27.6%)	18 (62.1%)	3 (10.3%)
		2	6 (22.2%)	19 (70.4%)	2 (7.4%)
		3	10 (29.4%)	20 (58.8%)	4 (11.8%)
Common	Llama-3.3-70B	1	31 (60.8%)	17 (33.3%)	3 (5.9%)
		2	27 (54.0%)	22 (44.0%)	1 (2.0%)
		3	27 (55.1%)	21 (42.9%)	1 (2.0%)
	Mistral-Small-3.1-24B	1	25 (46.3%)	25 (46.3%)	4 (7.4%)
		2	29 (50.0%)	25 (43.1%)	4 (6.9%)
		3	20 (37.0%)	29 (53.7%)	5 (9.3%)
DB2	Llama-3.3-70B	1	61 (91.0%)	6 (9.0%)	0 (0.0%)
		2	48 (88.9%)	6 (11.1%)	0 (0.0%)
		3	45 (86.5%)	7 (13.5%)	0 (0.0%)
	Mistral-Small-3.1-24B	1	18 (46.2%)	18 (46.2%)	3 (7.7%)
		2	15 (46.9%)	15 (46.9%)	2 (6.3%)
		3	14 (45.2%)	16 (51.6%)	1 (3.2%)
Log	Llama-3.3-70B	1	23 (79.3%)	4 (13.8%)	2 (6.9%)
		2	27 (90.0%)	2 (6.7%)	1 (3.3%)
		3	23 (100.0%)	0 (0.0%)	0 (0.0%)
	Mistral-Small-3.1-24B	1	4 (20.0%)	14 (70.0%)	2 (10.0%)
		2	3 (14.3%)	15 (71.4%)	3 (14.3%)
		3	3 (14.3%)	15 (71.4%)	3 (14.3%)

The pass rates in Table 4 show that repair improves results for both models in the majority of repositories. For Llama-3.3-70B, all four repositories benefit. ACL shows modest repair effectiveness (RE = 0.105–0.292) with an unchanged test count, so the gain comes from resolving initially failing tests. DB2 reaches RE = 0.478–0.731 while the executed test count grows moderately, from 182–199 to 209–216. Common, by contrast, yields RE values exceeding 1.0 in all three runs (1.078–1.780), with the executed test count rising from 192–214 to 238–295, so the additional passing tests exceed the number of initial failures. Log presents the most variable picture, including one negative RE value (Run 2: RE = −0.100) where fewer tests passed after repair.

For Mistral-Small-3.1-24B, repair improves pass rates in three of four repositories. ACL stands out with the highest RE values for this model (1.324–1.655), with the executed test count rising from 117–130 to 158–191. Common and DB2 show RE values near or above

Table 4 Test repair results on industrial repositories across three runs

Repo	Model	Run	Tests	Passed	Failed	RE
ACL	Llama-3.3-70B	1	319	273 (85.6%)	46 (14.4%)	0.292
		2	319	268 (84.0%)	51 (16.0%)	0.105
		3	340	283 (83.2%)	57 (16.8%)	0.174
	Mistral-Small-3.1-24B	1	191	149 (78.0%)	42 (22.0%)	1.655
		2	158	126 (79.7%)	32 (20.3%)	1.333
		3	174	132 (75.9%)	42 (24.1%)	1.324
Common	Llama-3.3-70B	1	279	218 (78.1%)	61 (21.9%)	1.078
		2	295	231 (78.3%)	64 (21.7%)	1.780
		3	238	204 (85.7%)	34 (14.3%)	1.184
	Mistral-Small-3.1-24B	1	214	160 (74.8%)	54 (25.2%)	0.889
		2	223	168 (75.3%)	55 (24.7%)	0.862
		3	229	173 (75.5%)	56 (24.5%)	1.148
DB2	Llama-3.3-70B	1	216	164 (75.9%)	52 (24.1%)	0.478
		2	209	166 (79.4%)	43 (20.6%)	0.611
		3	211	168 (79.6%)	43 (20.4%)	0.731
	Mistral-Small-3.1-24B	1	178	127 (71.3%)	51 (28.7%)	1.051
		2	148	109 (73.6%)	39 (26.4%)	0.625
		3	154	119 (77.3%)	35 (22.7%)	0.645
Log	Llama-3.3-70B	1	129	106 (82.2%)	23 (17.8%)	0.414
		2	136	100 (73.5%)	36 (26.5%)	−0.100
		3	133	114 (85.7%)	19 (14.3%)	0.304
	Mistral-Small-3.1-24B	1	60	35 (58.3%)	25 (41.7%)	−0.250
		2	70	43 (61.4%)	27 (38.6%)	0.048
		3	67	43 (64.2%)	24 (35.8%)	0.190

1.0 and a comparable growth in executed tests. The exception is Log, where RE values of −0.250 to 0.190 confirm that repair is largely counterproductive for the smaller model in this repository, with Run 1 exhibiting a negative RE that signals net regression.

The Log/Mistral Run 1 regression provides a concrete example of a negative RE value. Before repair, this run contained 60 tests, of which 40 passed and 20 failed. After repair, the total number of tests remained 60, but only 35 passed and 25 failed, yielding RE = −0.250. The corresponding error-category shift in Table 5 shows that `no_output` errors decreased only slightly, from 14 to 13, while `test_failure` errors increased from 4 to 10. Thus, repair did not substantially resolve the dominant structural failures in this run and instead increased semantic assertion failures. Because the total test count remained unchanged, this pattern suggests that the repair step modified tests in a way that made some previously passing or executable tests fail semantically, rather than preserving already passing behavior. Log/Llama Run 2 shows a related but slightly different pattern: the total number of tests increased from 133 to 136 while passing tests decreased from 103 to 100, suggesting that the negative RE value there may reflect both newly introduced failing tests and regressions among previously passing tests.

Error Type Rate (ETR) Table 5 shows the error types after automatic repair.

After repair, significant changes in the error distribution are evident, as captured by the shift values (Δ) in Table 5. For Llama-3.3-70B, `test_failure` errors still predominate

Table 5 Distribution of error types after repair on industrial repositories with error category shift (all runs)

			Post-Repair Counts			Shift (Δ)		
Repo	Model	Run	test_fail	no_out	unk	Δtf	Δno	Δunk
ACL	Llama-3.3-70B	1	43 (93.5%)	1 (2.2%)	2 (4.3%)	−22	+1	+2
		2	48 (94.1%)	0 (0.0%)	3 (5.9%)	−8	0	+2
		3	52 (91.2%)	1 (1.8%)	4 (7.0%)	−17	+1	+4
	Mistral-Small-3.1-24B	1	27 (64.3%)	10 (23.8%)	5 (11.9%)	+19	−8	+2
		2	16 (50.0%)	12 (37.5%)	4 (12.5%)	+10	−7	+2
		3	28 (66.7%)	10 (23.8%)	4 (9.5%)	+18	−10	0
Common	Llama-3.3-70B	1	57 (93.4%)	2 (3.3%)	2 (3.3%)	+26	−15	−1
		2	62 (96.9%)	1 (1.6%)	1 (1.6%)	+35	−21	0
		3	30 (88.2%)	2 (5.8%)	2 (5.9%)	+3	−19	+1
	Mistral-Small-3.1-24B	1	36 (66.7%)	15 (27.8%)	3 (5.6%)	+11	−10	−1
		2	37 (67.3%)	14 (25.5%)	4 (7.3%)	+8	−11	0
		3	34 (60.7%)	16 (28.6%)	6 (10.7%)	+14	−13	+1
DB2	Llama-3.3-70B	1	52 (100%)	0 (0.0%)	0 (0.0%)	−9	−6	0
		2	43 (100%)	0 (0.0%)	0 (0.0%)	−5	−6	0
		3	43 (100%)	0 (0.0%)	0 (0.0%)	−2	−7	0
	Mistral-Small-3.1-24B	1	35 (68.6%)	12 (23.5%)	4 (7.8%)	+17	−6	+1
		2	26 (66.7%)	11 (28.2%)	2 (5.1%)	+11	−4	0
		3	20 (57.1%)	13 (37.1%)	2 (5.7%)	+6	−3	+1
Log	Llama-3.3-70B	1	22 (95.7%)	1 (4.3%)	0 (0.0%)	−1	−3	−2
		2	33 (91.7%)	2 (5.6%)	1 (2.8%)	+6	0	0
		3	19 (100.0%)	0 (0.0%)	0 (0.0%)	−4	0	0
	Mistral-Small-3.1-24B	1	10 (40.0%)	13 (52.0%)	2 (8.0%)	+6	−1	0
		2	12 (44.4%)	13 (48.1%)	2 (7.4%)	+9	−2	−1
		3	11 (45.8%)	11 (45.8%)	2 (8.3%)	+8	−4	−1

(88%–100%), but the underlying dynamics differ by repository. In ACL, where test counts remain stable, `test_failure` counts decrease substantially (Δtf = −8 to −22), indicating genuine semantic repairs. In Common, by contrast, `test_failure` counts increase (Δtf = +3 to +35) while `no_output` errors drop sharply (Δno = −15 to −21), revealing that repair resolves structural problems while the tests that now execute include ones that fail semantically. DB2 presents the cleanest outcome: both error types decrease (Δtf = −2 to −9, Δno = −6 to −7), and all remaining errors after repair are regular test failures (100%), indicating very effective repair of syntactic problems.

For Mistral-Small-3.1-24B, the proportion of `no_output` errors remains notable after repair (24%–52%), but the shift values confirm consistent reductions across all repositories (Δno = −1 to −13). However, this improvement is accompanied by increases in `test_failure` counts across the board (Δtf = +6 to +19), suggesting that repair successfully brings tests to compilation but the smaller model struggles to produce semantically correct assertions. This pattern aligns with the high RE values observed for Mistral in Table 4, where the growth in executed tests raises both the total test count and the number of semantic failures. The `unknown` error types remain at 5%–13% after repair for Mistral and 0%–7% for Llama, showing no major change and indicating that this small proportion of unclassifiable errors is largely unaffected by the repair mechanism.

Table 6 Code coverage on industrial repositories (after repair, all modules)

Repository	Model	Run	Stmt	Bran	Cond	Sub
ACL - Utils::ACL						
	Llama-3.3-70B	1	61.3	47.7	29.4	88.3
		2	77.0	61.3	39.2	90.7
		3	62.4	52.2	37.2	86.0
	Mistral-Small-3.1-24B	1	52.4	34.0	27.4	55.8
		2	13.8	5.3	0.0	30.2
		3	37.8	24.2	1.9	48.8
ACL - Utils::ACL::Entry						
	Llama-3.3-70B	1	82.9	68.1	51.8	97.5
		2	83.8	67.6	51.8	100.0
		3	81.5	70.5	53.7	97.5
	Mistral-Small-3.1-24B	1	29.1	15.2	9.2	55.0
		2	13.2	0.4	0.0	42.5
		3	28.0	14.2	7.4	52.5
Common - Utils::Common						
	Llama-3.3-70B	1	74.1	52.8	43.4	82.8
		2	67.6	47.1	34.0	75.7
		3	67.2	47.7	37.6	77.1
	Mistral-Small-3.1-24B	1	43.5	28.2	18.8	60.0
		2	41.7	27.4	17.3	61.4
		3	41.9	26.2	23.1	60.0
DB2 - Utils::DB2						
	Llama-3.3-70B	1	85.9	63.8	53.9	91.6
		2	75.1	60.4	53.9	91.6
		3	85.1	64.5	50.7	91.6
	Mistral-Small-3.1-24B	1	36.9	18.7	25.4	66.6
		2	36.5	19.4	19.0	66.6
		3	39.3	29.1	15.8	66.6
DB2 - Utils::DB2::DiffTables						
	Llama-3.3-70B	1	40.5	36.3	66.6	90.0
		2	34.1	27.2	66.6	80.0
		3	34.1	27.2	66.6	80.0
	Mistral-Small-3.1-24B	1	40.5	36.3	66.6	90.0
		2	34.1	27.2	66.6	80.0
		3	34.1	27.2	66.6	80.0
DB2 - Utils::DB2::Measurements						
	Llama-3.3-70B	1	91.6	66.6	n/a	93.7
		2	91.6	66.6	n/a	93.7
		3	91.6	66.6	n/a	93.7
	Mistral-Small-3.1-24B	1	63.8	33.3	n/a	56.2
		2	55.5	0.0	n/a	50.0
		3	55.5	0.0	n/a	50.0
DB2 - Utils::DB2::Requests						
	Llama-3.3-70B	1	70.8	56.2	n/a	93.7
		2	75.0	62.5	n/a	93.7
		3	76.3	62.5	n/a	93.7
	Mistral-Small-3.1-24B	1	30.5	18.7	n/a	50.0
		2	31.9	18.7	n/a	50.0

Table 6 (continued)

Repository	Model	Run	Stmt	Bran	Cond	Sub
		3	34.7	18.7	n/a	56.2
Log - Utils::Log						
	Llama-3.3-70B	1	74.2	50.0	35.1	95.2
		2	74.7	53.3	44.4	95.2
		3	77.2	53.3	38.8	97.6
	Mistral-Small-3.1-24B	1	19.9	4.3	0.0	50.0
		2	23.5	6.7	0.0	52.3
		3	25.2	6.7	0.0	54.7

After having created the initial test suite and thereafter run the repairs, we calculate the code coverage.

4.2.3 Code coverage (RQ3) - industrial systems

Table 6 shows the coverage metrics for all modules across all three runs after repair.

The coverage results show consistent patterns across both models and coverage types. A coverage hierarchy emerges: subroutine coverage is consistently highest, followed by statement coverage. For both models, branch coverage follows and condition coverage is lowest, with a few exceptions in the DB2 modules.

Llama-3.3-70B achieves substantially higher coverage than Mistral-Small-3.1-24B across all modules except DB2::DiffTables, where both models produce identical results across all coverage types and runs (e.g., statement 34.1%–40.5%, condition 66.6%). Llama also demonstrates higher consistency across runs, with several modules showing low variance: DB2::Measurements achieves 91.6% statement, 66.6% branch, and 93.7% subroutine coverage identically in all three runs, and DB2 subroutine coverage remains constant at 91.6%. The main exception is the ACL module, where Run 2 reaches 77.0% statement and 61.3% branch coverage while Runs 1 and 3 remain near 62% and 48%–52% respectively. ACL::Entry, by contrast, is highly stable across all coverage types (statement 81.5%–83.8%, branch 67.6%–70.5%, condition 51.8%–53.7%).

Mistral-Small-3.1-24B exhibits considerably greater run-to-run variability, most notably for the ACL modules: Utils::ACL statement coverage ranges from 13.8% to 52.4%, with branch coverage dropping as low as 5.3% in Run 2. For Common, DB2, and Log, Mistral shows more stable but consistently low values: Common statement coverage at 41.7%–43.5% with branch coverage of 26.2%–28.2%, DB2 at 36.5%–39.3% statement with subroutine coverage constant at 66.6%, and Log at 19.9%–25.2% statement coverage. The disparity between models is particularly pronounced for condition coverage, where Mistral achieves zero in several modules (ACL Run 2, ACL::Entry Run 2, and all three Log runs), whereas Llama consistently reaches 29%–54% in the same modules. Similarly, Mistral's branch coverage for DB2::Measurements drops to 0.0% in Runs 2 and 3, compared to Llama's stable 66.6%.

Table 7 Initial test generation on open-source distributions across three runs

Repository	Model	Run	Tests	Passed	Failed
XML::Simple	Llama-3.3-70B	1	228	178 (78.1%)	50 (21.9%)
		2	213	157 (73.7%)	56 (26.3%)
		3	216	164 (75.9%)	52 (24.1%)
	Mistral-Small-3.1-24B	1	108	50 (46.3%)	58 (53.7%)
		2	109	51 (46.8%)	58 (53.2%)
		3	113	53 (46.9%)	60 (53.1%)
Tie::IxHash	Llama-3.3-70B	1	139	111 (79.9%)	28 (20.1%)
		2	124	101 (81.5%)	23 (18.5%)
		3	138	112 (81.2%)	26 (18.8%)
	Mistral-Small-3.1-24B	1	112	63 (56.2%)	49 (43.8%)
		2	133	76 (57.1%)	57 (42.9%)
		3	111	63 (56.8%)	48 (43.2%)
Path::Class	Llama-3.3-70B	1	305	230 (75.4%)	75 (24.6%)
		2	320	247 (77.2%)	73 (22.8%)
		3	317	248 (78.2%)	69 (21.8%)
	Mistral-Small-3.1-24B	1	149	80 (53.7%)	69 (46.3%)
		2	150	81 (54.0%)	69 (46.0%)
		3	152	84 (55.3%)	68 (44.7%)

4.2.4 Results on open-source systems

The same configuration was applied to the three CPAN distributions described in Section 4.1.3. A pooled comparison of both subject sets is given in Table 12.

Test Pass Rate (TPR) Table 7 shows the initial generation results.

For Llama-3.3-70B, the initial pass rates lie between 73.7% and 81.5%. Tie::IxHash is the most stable of the three (79.9%–81.5%). XML::Simple varies more in both test count (213–228) and pass rate (73.7%–78.1%). Path::Class produces the largest suites (305–320 tests) with pass rates of 75.4%–78.2%.

For Mistral-Small-3.1-24B, the pass rates are lower and more tightly grouped within each distribution: XML::Simple at 46.3%–46.9%, Tie::IxHash at 56.2%–57.1%, and Path::Class

Table 8 Distribution of error types in initial generation on open-source distributions (all runs)

Repo	Model	Run	test_failure	no_output	unknown
XML::Simple	Llama-3.3-70B	1	36 (72.0%)	1 (2.0%)	13 (26.0%)
		2	43 (76.8%)	0 (0.0%)	13 (23.2%)
		3	35 (67.3%)	1 (1.9%)	16 (30.8%)
	Mistral-Small-3.1-24B	1	18 (31.0%)	18 (31.0%)	22 (37.9%)
		2	18 (31.0%)	17 (29.3%)	23 (39.7%)
		3	20 (33.3%)	16 (26.7%)	24 (40.0%)
Tie::IxHash	Llama-3.3-70B	1	22 (78.6%)	0 (0.0%)	6 (21.4%)
		2	15 (65.2%)	1 (4.3%)	7 (30.4%)
		3	20 (76.9%)	0 (0.0%)	6 (23.1%)
	Mistral-Small-3.1-24B	1	29 (59.2%)	4 (8.2%)	16 (32.7%)
		2	36 (63.2%)	5 (8.8%)	16 (28.1%)
		3	25 (52.1%)	5 (10.4%)	18 (37.5%)
Path::Class	Llama-3.3-70B	1	54 (72.0%)	5 (6.7%)	16 (21.3%)
		2	64 (87.7%)	2 (2.7%)	7 (9.6%)
		3	54 (78.3%)	3 (4.3%)	12 (17.4%)
	Mistral-Small-3.1-24B	1	6 (8.7%)	17 (24.6%)	46 (66.7%)
		2	7 (10.1%)	16 (23.2%)	46 (66.7%)
		3	5 (7.4%)	14 (20.6%)	49 (72.1%)

at 53.7%–55.3%. As in the industrial set, Mistral also generates fewer executable tests than Llama.

Error Type Rate (ETR) Table 8 shows the distribution of error types for all three runs of initial generation.

The error distribution again differs clearly between the models. For Llama-3.3-70B, `test_failure` dominates (65%–88% of errors). Structural `no_output` errors stay rare

(0%–7%). The remaining failures are mostly `unknown` (9.6%–30.8%), which is a larger share than in the industrial set.

For Mistral-Small-3.1-24B, `unknown` is much more frequent, especially on Path::Class (67%–72%). XML::Simple splits more evenly across the three categories (31.0%–33.3% `test_failure`, 26.7%–31.0% `no_output`, 37.9%–40.0% `unknown`). On Tie::IxHash, `test_failure` is the largest class (52%–63%) and `no_output` the smallest (8.2%–10.4%).

Test Pass Rate (TPR) Table 9 shows the results after repair.

Repair improves the number of passing tests in all 18 open-source runs. For Llama-3.3-70B, Tie::IxHash again shows the strongest outcome (RE = 0.500–0.870; pass rates 85.0%–89.0%). Path::Class improves more modestly (RE = 0.233–0.427). XML::Simple Run 1 is a case where RE is positive (0.440) while the pass *rate* falls from 78.1% to 76.6%, because 22 more tests pass while the executed test count grows by 33. This is the same pattern already seen for Log in the industrial set, where Runs 2 and 3 of Mistral-Small-3.1-24B also combine a positive RE with a falling pass rate.

For Mistral-Small-3.1-24B, RE is positive in every run and exceeds 1.0 for Path::Class (1.391–1.559). Those values come from a large increase in the number of executed tests (149–152 initially, 310–333 after repair), not from a large rise in the pass rate, which remains in the mid-50% range. XML::Simple stays the weakest of the three for the smaller model (52.9%–56.4% after repair).

Error Type Rate (ETR) Table 10 shows the error types after automatic repair.

After repair, Llama-3.3-70B still has mostly `test_failure` errors (75%–90%). On Tie::IxHash and Path::Class the counts fall or stay nearly stable (Δtf = −10 to +0). On

Table 9 Test repair results on open-source distributions across three runs

Repo	Model	Run	Tests	Passed	Failed	RE
XML::Simple	Llama-3.3-70B	1	261	200 (76.6%)	61 (23.4%)	0.440
		2	226	172 (76.1%)	54 (23.9%)	0.268
		3	242	187 (77.3%)	55 (22.7%)	0.442
	Mistral-Small-3.1-24B	1	156	88 (56.4%)	68 (43.6%)	0.655
		2	155	86 (55.5%)	69 (44.5%)	0.603
		3	189	100 (52.9%)	89 (47.1%)	0.783
Tie::IxHash	Llama-3.3-70B	1	146	130 (89.0%)	16 (11.0%)	0.679
		2	141	121 (85.8%)	20 (14.2%)	0.870
		3	147	125 (85.0%)	22 (15.0%)	0.500
	Mistral-Small-3.1-24B	1	163	103 (63.2%)	60 (36.8%)	0.816
		2	189	117 (61.9%)	72 (38.1%)	0.719
		3	174	105 (60.3%)	69 (39.7%)	0.875
Path::Class	Llama-3.3-70B	1	326	262 (80.4%)	64 (19.6%)	0.427
		2	332	264 (79.5%)	68 (20.5%)	0.233
		3	334	273 (81.7%)	61 (18.3%)	0.362
	Mistral-Small-3.1-24B	1	310	176 (56.8%)	134 (43.2%)	1.391
		2	333	185 (55.6%)	148 (44.4%)	1.507
		3	328	190 (57.9%)	138 (42.1%)	1.559

Table 10 Distribution of error types after repair on open-source distributions with error category shift (all runs)

			Post-Repair Counts			Shift (Δ)		
Repo	Model	Run	test_fail	no_out	unk	Δtf	Δno	Δunk
XML::Simple	Llama-3.3-70B	1	55 (90.2%)	1 (1.6%)	5 (8.2%)	+19	0	−8
		2	45 (83.3%)	1 (1.9%)	8 (14.8%)	+2	+1	−5
		3	48 (87.3%)	0 (0.0%)	7 (12.7%)	+13	−1	−9
	Mistral-Small-3.1-24B	1	35 (51.5%)	13 (19.1%)	20 (29.4%)	+17	−5	−2
		2	39 (56.5%)	11 (15.9%)	19 (27.5%)	+21	−6	−4
		3	61 (68.5%)	10 (11.2%)	18 (20.2%)	+41	−6	−6
Tie::IxHash	Llama-3.3-70B	1	12 (75.0%)	0 (0.0%)	4 (25.0%)	−10	0	−2
		2	15 (75.0%)	1 (5.0%)	4 (20.0%)	0	0	−3
		3	19 (86.4%)	0 (0.0%)	3 (13.6%)	−1	0	−3
	Mistral-Small-3.1-24B	1	46 (76.7%)	2 (3.3%)	12 (20.0%)	+17	−2	−4
		2	57 (79.2%)	2 (2.8%)	13 (18.1%)	+21	−3	−3
		3	54 (78.3%)	3 (4.3%)	12 (17.4%)	+29	−2	−6
Path::Class	Llama-3.3-70B	1	53 (82.8%)	3 (4.7%)	8 (12.5%)	−1	−2	−8
		2	61 (89.7%)	2 (2.9%)	5 (7.4%)	−3	0	−2
		3	54 (88.5%)	2 (3.3%)	5 (8.2%)	0	−1	−7
	Mistral-Small-3.1-24B	1	92 (68.7%)	15 (11.2%)	27 (20.1%)	+86	−2	−19
		2	109 (73.6%)	14 (9.5%)	25 (16.9%)	+102	−2	−21
		3	100 (72.5%)	13 (9.4%)	25 (18.1%)	+95	−1	−24

XML::Simple they rise (Δtf = +2 to +19) while `unknown` decreases, which matches the extra tests added during repair. `no_output` remains close to zero.

For Mistral-Small-3.1-24B, `no_output` and `unknown` decrease in every run, but `test_failure` increases throughout (Δtf = +17 to +102). Path::Class shows this most clearly: repair more than doubles the suite and converts many previously unclassifiable failures into executable tests that then fail assertions. This is the same structural-to-semantic shift already observed for Mistral on the industrial repositories.

Execution stability While commenting out the failing tests to measure coverage, we noticed that the number of passing tests varied between runs of the same files, indicating non-

Table 11 Code coverage on open-source distributions (after repair, all modules)

Repository	Model	Run	Stmt	Bran	Cond	Sub
XML::Simple - XML::Simple						
	Llama-3.3-70B	1	67.5	50.4	42.0	92.0
		2	65.5	50.0	43.6	90.0
		3	67.5	47.7	47.6	94.0
	Mistral-Small-3.1-24B	1	22.9	14.1	13.4	38.0
		2	17.5	8.6	13.4	42.0
		3	22.0	10.6	13.4	48.0
Tie::IxHash - Tie::IxHash						
	Llama-3.3-70B	1	88.9	76.6	83.3	87.1
		2	87.1	78.3	83.3	90.3
		3	88.0	75.0	50.0	87.1
	Mistral-Small-3.1-24B	1	40.2	33.3	83.3	54.8
		2	49.1	61.6	83.3	61.2
		3	37.1	33.3	83.3	48.3
Path::Class - Path::Class						
	Llama-3.3-70B	1	100.0	n/a	n/a	100.0
		2	95.6	n/a	n/a	90.9
		3	95.6	n/a	n/a	90.9
	Mistral-Small-3.1-24B	1	95.6	n/a	n/a	90.9
		2	95.6	n/a	n/a	90.9
		3	100.0	n/a	n/a	100.0
Path::Class - Path::Class::Dir						
	Llama-3.3-70B	1	91.8	77.0	67.5	90.0
		2	92.8	78.1	70.2	92.5
		3	91.8	79.1	70.2	90.0
	Mistral-Small-3.1-24B	1	52.0	32.2	27.0	62.5
		2	45.4	25.0	29.7	57.5
		3	40.3	23.9	29.7	55.0
Path::Class - Path::Class::Entity						
	Llama-3.3-70B	1	90.3	78.5	100.0	85.7
		2	94.2	85.7	100.0	90.4
		3	94.2	78.5	100.0	90.4
	Mistral-Small-3.1-24B	1	57.6	42.8	66.6	57.1
		2	57.6	42.8	66.6	57.1
		3	57.6	42.8	66.6	57.1
Path::Class - Path::Class::File						
	Llama-3.3-70B	1	90.0	72.0	30.0	93.1
		2	90.0	75.0	40.0	93.1
		3	88.4	75.0	40.0	89.6
	Mistral-Small-3.1-24B	1	53.7	39.7	0.0	62.0
		2	57.8	45.5	0.0	68.9
		3	57.0	42.6	0.0	65.5

deterministic assertions. Running each archived stage five times in the same environment resulted in different outcomes for only 10 out of 10,166 tests (0.098%). All of these inconsistencies occurred in two test files whose assertions depend on the order in which Perl

traverses a hash. At this level, the observed execution flakiness has no effect on the reported pass rates or coverage results.

Code coverage Table 11 shows coverage after repair. Path::Class contributes four modules, so the table has six module blocks.

The coverage hierarchy from the industrial set is largely visible here as well: subroutine coverage is highest, followed by statement coverage. For Llama-3.3-70B, branch and condition coverage follow in that order except on `Tie::IxHash` and `Path::Class::Entity`, whereas for Mistral-Small-3.1-24B condition coverage exceeds branch coverage on all modules for which both are defined except `Path::Class::File`. Llama-3.3-70B reaches statement coverage of 65.5%–100.0%. Tie::IxHash and the Path::Class modules sit at the upper end (87.1%–100.0% statement). XML::Simple is lower (65.5%–67.5% statement, 47.7%–50.4% branch), which matches its more demanding option space. Path::Class itself has no branches or conditions, and both models cover this small facade at 95.6%–100.0%.

Mistral-Small-3.1-24B remains well below Llama on the larger modules. XML::Simple statement coverage is 17.5%–22.9%. Path::Class::Dir ranges from 40.3% to 52.0%, and Path::Class::File has 0.0% condition coverage in all three runs. Tie::IxHash is the relative exception for the smaller model (37.1%–49.1% statement; condition coverage 83.3% in all runs).

Comparison of the two subject sets Table 12 pools all runs of each subject set. Pass rates and RE are computed from the pooled test counts, using the same RE definition as in (1). Error shares are proportions of the pooled initial-generation failures.

Table 12 Pooled comparison of industrial and open-source subjects

Metric	Model	Industrial	Open-source
Initial pass rate	Llama-3.3-70B	76.5%	77.4%
	Mistral-Small-3.1-24B	70.7%	52.9%
Post-repair pass rate	Llama-3.3-70B	81.3%	80.5%
	Mistral-Small-3.1-24B	74.2%	57.6%
Repair effectiveness (RE)	Llama-3.3-70B	0.601	0.412
	Mistral-Small-3.1-24B	0.881	1.024
Statement coverage	Llama-3.3-70B	34.1–91.6%	65.5–100.0%
	Mistral-Small-3.1-24B	13.2–63.8%	17.5–100.0%
`test_failure` share (initial)	Llama-3.3-70B	84.2%	75.9%
	Mistral-Small-3.1-24B	36.9%	30.6%
`no_output` share (initial)	Llama-3.3-70B	14.3%	2.9%
	Mistral-Small-3.1-24B	54.5%	20.9%
`unknown` share (initial)	Llama-3.3-70B	1.5%	21.2%
	Mistral-Small-3.1-24B	8.6%	48.5%

Table 13 Coverage of the generated suites and of the distributions' own test suites

	Own suite (CPAN Cover)				Llama-3.3-70B			
Module	Stmt	Bran	Cond	Sub	Stmt	Bran	Cond	Sub
`XML::Simple`	94.6	89.8	89.6	94.0	66.8	49.4	44.4	92.0
`Tie::IxHash`	71.6	66.6	33.3	74.1	88.0	76.6	72.2	88.2
`Path::Class`	100.0	n/a	n/a	100.0	97.1	n/a	n/a	93.9
`Path::Class::Dir`	97.4	86.4	67.5	95.0	92.1	78.1	69.3	90.8
`Path::Class::Entity`	98.0	78.5	66.6	95.2	92.9	80.9	100.0	88.8
`Path::Class::File`	86.7	57.3	60.0	82.7	89.5	74.0	36.7	91.9

For Llama-3.3-70B, the two subject sets are close. The initial pass rate is 76.5% on the industrial repositories and 77.4% on the open-source ones. After repair the values are 81.3% and 80.5%. RE is higher on the industrial side (0.601 vs. 0.412): repair yields 358 additional passing tests against 596 initial failures, compared with 186 against 452 on the open-source distributions. `test_failure` remains the dominant error type for Llama on both, at 84.2% industrial and 75.9% open-source, while `no_output` errors are more frequent on the industrial side (14.3% vs. 2.9%) and `unknown` errors rarer (1.5% vs. 21.2%).

For Mistral-Small-3.1-24B, the gap between the two subject sets is larger. The initial pass rate is 70.7% on the industrial repositories and 52.9% on the open-source ones. After repair the values are 74.2% and 57.6%. The high RE of 1.024 again reflects the growth in executed tests, especially on Path::Class, rather than a high pass rate. On both subject sets, Mistral has more structural and unclassifiable failures than Llama, but the balance differs: `no_output` accounts for 54.5% of the industrial failures against 20.9% of the open-source ones, while `unknown` accounts for 8.6% against 48.5%.

Reference to the existing test suites Since all three subjects are public, the coverage produced by their own test suites is available as well. Table 13 presents these results alongside the generated suites. For Llama-3.3-70B, the generated tests achieve higher coverage than the existing suite for `Tie::IxHash` across all four coverage metrics, and for `Path::Class::File` in statement, branch and subroutine coverage. For the other `Path::Class` modules they remain within six percentage points in statement coverage, and `XML::Simple` is the only case where they fall clearly behind. Averaged over the modules, subroutine coverage shows the smallest difference, including 92.0% against 94.0% for `XML::Simple`, which is consistent with generated tests calling most of the available functions while covering fewer of the internal execution paths. The reference measurements were obtained through CPAN's coverage service (Johnson, 2026), which ran the suites on older Perl versions, from 5.20.0 to 5.26.1, and in a different environment. They therefore indicate the overall coverage level rather than a strictly controlled comparison.

Taken together, the open-source runs show that the workflow is not specific to IBM's codebase or development environment. The larger model reaches comparable pass rates on industrial and open-source Perl. The smaller model remains weaker in both settings, and more so on the open-source systems. The prompts, generated tests, and result files for the three CPAN distributions are published in full in the replication package.

5 Discussion

We first compare the strengths and weaknesses of our approach with the findings of the related work. Thereafter, the key implications of the work are highlighted.

5.1 Comparison with related work

The evaluation shows that the developed approach for LLM-based test generation for legacy Perl code is fundamentally functional and delivers good results in several aspects when compared with the literature.

High executability rates Llama-3.3-70B achieves initial pass rates of 66%–82% on the industrial repositories, increasing to 74%–86% after repair, with improvements across all four repositories (mean gains of +1.8 to +8.7 percentage points (PP)). On the CPAN distributions the corresponding values are 74%–81% and 76%–89%. These results compare favorably with related work: Tang et al. (2024) report 69.6% executability for ChatGPT, Lops et al. (2025) achieve 80.85% compilation but only 30.85% pass rate for GPT-4, and the HITS approach by Wang et al. (2024b) reaches 69.11% for complex methods.

Good code coverage Llama-3.3-70B achieves statement coverage of 34.1%–91.6% across the industrial modules and 65.5%–100.0% across the CPAN modules, which is competitive with or exceeds related results: 55.4% for ChatGPT (Tang et al., 2024) and 59.1% line coverage for ChatGPT-3.5 (Karanjai et al., 2024), though Wang et al. (2024a) note that reported coverage varies widely across benchmarks (2%–89%). A closer baseline is available for the open-source subjects, whose own test suites reach statement coverage of 71.6%–100.0% across the six modules. Llama-3.3-70B exceeds this on `Tie::IxHash` and `Path::Class::File` and remains within six percentage points on three further modules, but falls clearly behind on `XML::Simple` (66.8% against 94.6%).

Despite the positive results, clear weaknesses of the approach also emerge. We also compare these findings with the related work.

Variable repair effectiveness Measured by pass rates (Table 4), automatic repair improves results for both models in most of the industrial repositories. For Llama-3.3-70B, mean pass rates increase across all four repositories, with DB2 showing the largest gain (+8.7 PP) and Log the smallest (+1.8 PP). However, post-repair consistency varies considerably, with Log exhibiting the highest variability ($s = 6.3$, range 73.5%–85.7%). Such non-determinism aligns with findings by Jalil et al. (2023), who observed that ChatGPT answer correctness varied for 9.7% of repeated identical queries.

For Mistral-Small-3.1-24B, repair improves three of four repositories but degrades Log by −4.8 PP. The error-category shifts make this failure mode more specific: for Log/Mistral, repair only marginally reduces structural `no_output` errors while increasing semantic `test_failure` counts. In Run 1, for example, `no_output` decreases from 14 to 13, whereas `test_failure` increases from 4 to 10. This suggests that the repair step sometimes transforms structurally invalid or previously passing tests into executable but

semantically incorrect tests, instead of preserving already passing behavior. On the CPAN distributions repair works more evenly. The pass rate rises in 17 of the 18 runs and falls only once, by 1.4 PP for XML::Simple with Llama-3.3-70B. This context-dependence of repair effectiveness is consistent with the literature: In the overview by Wang et al. (2024a), it is mentioned that during a study on the QuixBugs dataset, a success rate of 39 out of 40 errors was achieved when repairing Python bugs with Codex, yet this would never happen in real-world environments. A study mentioned there achieved success with only 16 out of 72 errors when repairing more complex deep learning programs from StackOverflow with ChatGPT. This confirms that LLM performance drastically decreases when problems gain complexity.

Model-dependent results The clear differences between Llama-3.3-70B and Mistral-Small-3.1-24B show a strong dependency of result quality on the model used. Mistral-Small-3.1-24B not only generates fewer executable tests but also achieves significantly lower coverage values. The gap is wider on the CPAN distributions, where the pooled initial pass rates are 77.4% and 52.9%, against 76.5% and 70.7% on the industrial repositories. Particularly problematic is the high proportion of `no_output` errors with Mistral-Small-3.1-24B, indicating general difficulties in generating executable test files. This observation is consistent with the findings of Yang et al. (2024), who showed that larger models like Phind-CodeLlama-34B-v2 and DeepSeekCoder-33B-Instruct achieved higher code coverage than smaller models. However, because the prompt templates were developed with Llama-3.3-70B, these differences should not be interpreted as model-capacity effects alone. Part of Mistral-Small-3.1-24B's lower performance may also reflect a weaker fit between the Llama-developed prompt interface and Mistral's response behavior.

Insufficient branch and condition coverage Despite good statement coverage (34.1%–91.6% for Llama-3.3-70B), branch coverage (27.2%–70.5%) and especially condition coverage (29.4%–66.6%) lag significantly behind. For Mistral-Small-3.1-24B, the difference is even greater: statement coverage of 13.2%–63.8% contrasts with branch coverage of 0.0%–36.3% and condition coverage of 0.0%–66.6%. The same pattern appears on the CPAN distributions, where Llama-3.3-70B reaches statement coverage of 65.5%–100.0% but branch coverage of only 47.7%–85.7%. This indicates that while the generated tests execute many lines of code, they do not cover all branches and edge cases. For quality assurance, more targeted prompts for generating edge-case tests would be necessary. Yang et al. (2024) compared LLMs with the traditional tool EvoSuite and found that even GPT-4 with a line coverage of 40.43% and a branch coverage of 31.78% fell far behind EvoSuite (78.91% and 76.59% respectively).

no_output problem A high proportion of failed tests falls into the `no_output` category, meaning these test files could not be executed at all. The causes are varied: incorrect module paths, syntactic errors that prevent Perl parsing, or false assumptions about the environment. This error class is particularly problematic because it cannot be fixed by better assertions but rather reveals general problems in the test structure itself. For Llama-3.3-70B, the proportion of these errors decreases through repair from 0.0-44.0% to 0.0-5.8%, for Mistral-Small-3.1-24B from 43.1-71.4% to 23.5-52.0% – an improvement, but still high. On the CPAN distributions this error type is much less common for Mistral-Small-3.1-24B, at 20.9% of the initial failures against 54.5% on the industrial repositories. Yang et al. (2024) report that

depending on model and configuration, between 34.44% and 61.78% of tests generated by LLMs are syntactically invalid. Lops et al. (2025) found that with GPT-4, approximately 19% of the generated test classes did not compile.

5.2 Defensive wrapper approach: a design trade-off

The systematic use of eval wrappers in generated tests represents a deliberate design decision with both advantages and limitations that merit discussion.

The primary motivation for the eval wrapper approach stems from the requirements of integration into CI/CD pipelines. In continuous integration environments, pipeline stability is paramount: a test suite that crashes mid-execution provides incomplete feedback and blocks deployment processes. The eval wrapper ensures that:

- **Pipeline stability**: A crash in the code under test does not abort the run, so the pipeline still produces a report covering every test file. This is critical for industrial CI/CD pipelines where hard crashes are unacceptable.
- **Comprehensive feedback**: Results are captured for the assertions that execute, enabling developers to see the scope of issues rather than only the first failure.
- **Preservation of potentially valuable tests**: Tests that fail due to runtime errors are retained rather than discarded, as their failure behavior may itself reveal bugs in the code under test.

The eval wrapper approach is not positioned as superior to syntax repair but as a *complementary safety layer*. While repair mechanisms optimize for code quality, our wrapper provides "last-mile resilience" ensuring that even tests which resist automated repair or cause unexpected runtime failures are handled gracefully rather than destabilizing the test runner.

A potential criticism of the wrapper approach concerns coverage metrics. Since Devel::Cover measures executed statements rather than assertion success, defensive execution could theoretically inflate coverage by running code paths that would otherwise abort. However, this concern is addressed through the system design:

1. **Comment-failures mode:** When coverage is measured, the system first comments out all failing tests. This ensures that only passing tests contribute to coverage metrics, eliminating any potential inflation from tests that execute code but fail assertions.
2. **Meaningful assertions**: Unlike Nessie, which generates tests without meaningful assertions, our approach produces tests with explicit verification of expected behavior.
3. **Fail behavior is informative**: A test that executes code and then fails an assertion still provides valuable information about which code paths are reachable and what the code actually does versus what was expected.

To our knowledge, this is the first work to systematically combine defensive eval wrappers with LLM-generated tests. While Nessie employs similar try/catch isolation, it does not use LLMs and has been criticized for lacking meaningful assertions. The combination of:

- LLM-generated tests with explicit assertions
- Defensive wrapper for runtime resilience

- Coverage measurement with failing tests commented out

represents a novel methodological contribution that addresses the specific challenges of LLM-generated code in automated pipelines.

Scope of the defensive pattern The wrapper protects calls to the code under test, but not the setup code that a test file requires around them. In the open-source subjects, where the generated tests can be inspected, most files that terminated before reporting any assertion did so because of faults in mock configuration. Although the generation prompt template instructs the models to mock external dependencies, it does not demonstrate this setup inside a wrapped call. The repair template does show the pattern, which is one reason `no_output` errors fall after repair. The models differ clearly in this respect: Llama-3.3-70B applied the mocking interface correctly in most cases, whereas Mistral-Small-3.1-24B did so far less reliably, which accounts for a substantial part of its higher `no_output` rate. The effectiveness of defensive execution therefore depends both on the pattern being specified in the prompt and on the model applying it consistently.

5.3 Practical implications and reflections

The present work shows several insights that are relevant beyond the Perl use case.

Effectiveness of multi-stage prompt engineering The developed approach first analyzes business logic, then generates tests, followed by error analysis and repair. In the evaluated workflow, this structure was useful for organizing the test-generation task and for making implicit domain assumptions explicit before assertions were formulated. However, because no single-phase baseline was evaluated, the results should be interpreted as evidence for the complete workflow rather than as proof that the business-logic phase independently improves test quality. Business logic analysis serves as a connection between domain knowledge often contained in code and the test assertions to be formulated, while the error analysis phase provides structured feedback for repair.

Iterative improvement through error feedback The mixed results of automatic repair show that simply feeding back error messages is not sufficient. Effective repair mechanisms require structured error diagnostics and targeted prompts that guide the LLM on what type of change should be made. Future research should investigate how error classification and repair strategies can be combined to increase success rates. In particular, repair workflows should separate already passing tests from failing tests before invoking the LLM, preserve passing tests unchanged, and provide only failing tests with their diagnostics for repair.

Perl as a test case for LLM generalization The fact that LLMs with probably minimal Perl-specific training (compared to Java or Python) can nevertheless generate functional tests suggests that underlying programming principles and test patterns are learned across languages. This is an important finding for applying LLMs to more niche languages or domain-specific languages.

The results of this work have implications for the practical use of LLM-based testing tools in software development.

GenAI as productivity enhancer For Llama-3.3-70B, pass rates reach 66%–82% for initial generation and 74%–86% after repair on the industrial systems, and 74%–81% and 76%–89% on the open-source ones. Together with mean statement coverage of 72% on the industrial systems and 88% on the open-source ones, this shows that LLMs can already serve as effective assistants for developers today. Instead of writing tests completely manually, developers can use LLM-generated tests as a starting point, review them, adapt and extend them. Repair mechanisms can provide a first step towards error correction but do not replace human review.

Qualitative value of generated tests Beyond quantitative metrics, the generated tests show practical value in the type of behavior they capture. Inspection of generated test samples across all seven subjects showed that they did not merely instantiate modules or invoke functions without assertions, but checked concrete program behavior like expected outputs, error cases, and relevant input variations. The Calculator example illustrates this with logic-oriented assertions and edge cases such as negative numbers, decimal values, zero, and non-numeric inputs. For the open-source subjects the coverage figures support this. For Llama-3.3-70B, the generated suites reach or come within six percentage points of the distributions' own test suites on five of six modules (Table 13). For legacy systems, this value is not limited to documenting intended functionality. When complete specifications, documentation, or existing test suites are unavailable, generated tests can also characterize the system's currently observed behavior. In this role, they can provide a regression-testing basis that helps developers detect unintended behavioral changes during later refactoring, maintenance, or extension.

Not deployable as complete replacement The clear differences between models, problems with automatic repair, and insufficient branch/condition coverage show that LLMs in their current state cannot serve as a complete replacement for human test engineers.

Cost-benefit trade-off For legacy systems with existing but incomplete test coverage, the use of LLM-based test generation can nevertheless be economically sensible. Even at the lowest pass rate after repair, 74%, and moderate coverage, manual effort is greatly reduced as the generated tests serve as a basis. The work of Schäfer et al. (2024) shows that LLM-generated tests are often more maintainable in practice than traditionally auto-generated tests because they are more human-readable and understandable.

CI/CD integration and pipeline stability A key practical contribution of this work is demonstrating how LLM-based test generation can be integrated into CI/CD pipelines. The defensive eval wrapper approach ensures that generated test suites never crash the pipeline, even when individual tests fail or encounter unexpected runtime errors. This is critical for industrial adoption, where pipeline stability is non-negotiable. The git-diff mode (Section 3.5.3) enables incremental test generation: when code changes are committed, only the affected functions receive new or updated tests. This maintains test coverage as the codebase evolves

without requiring full regeneration, making the approach sustainable for continuous development workflows.

Handling imperfect tests The approach demonstrates that LLM-generated tests with runtime errors need not be discarded. Unlike approaches that delete failing tests or invest in repair loops, our wrapper-based strategy retains all tests and captures their failure behavior. This has practical value: a test that crashes may reveal that the code under test has unexpected behavior (e.g., missing null checks, unhandled edge cases). Automated error commenting (the comment-failures mode) allows teams to accumulate a backlog of known issues for later manual review, rather than losing this information.

Significant model dependency The choice of LLM model has a decisive influence on result quality. Larger models (70B parameters) generate far more tests and achieve roughly twice the statement coverage of smaller models (24B parameters) on the industrial repositories, and about 1.6 times on the CPAN distributions. This shows the importance of model selection for practical applications and suggests that the complexity of the test generation task requires substantial model capacity.

6 Conclusion

This work investigates the use of Large Language Models (LLMs) for automated generation of unit tests for legacy Perl code, a programming language that has been largely neglected in previous research on LLM-based test generation. Motivated by missing test coverage, time- and resource-intensive manual test creation, and the limitations of traditional automated approaches, the work develops a modular workflow that combines static code analysis, structured prompt engineering, and automatic test repair. We formulated three research questions.

RQ1 (related to O1) *To what extent can LLM-based approaches automatically generate unit tests for legacy Perl modules that are syntactically correct and can be executed in a real test environment?* LLM-based approaches with modern, powerful models (70B+ parameters) and repair mechanisms can generate syntactically correct and executable unit tests for legacy Perl code, maintaining approximately a maximum rate of 5.8% `no_output` errors on the industrial repositories and 5.0% on the CPAN distributions. Success rate depends on the model used, with larger models delivering significantly better results.

RQ2 (related to O2) *How effective are repair mechanisms for solving problems?* Measured by pass rates, repair mechanisms are effective for the majority of repository–model combinations. For Llama-3.3-70B, mean pass rates improve in all four repositories (+1.8 to +8.7 PP), for Mistral-Small-3.1-24B, in three of four (+1.2 to +7.9 PP). Repair is especially effective at fixing structural `no_output` errors (Llama: 0–44% reduced to 0–6%; Mistral: 43–71% reduced to 24–52%). However, repair introduces increased variability across runs, and for the Log/Mistral combination, it degrades performance (−4.8 PP). On the CPAN distributions, mean pass rates improve in all three for both models (Llama-3.3-70B +0.8 to +5.8 PP, Mistral-Small-3.1-24B +2.4 to +8.3 PP). `no_output` errors fall from 0–7%

to 0–5% for Llama-3.3-70B and from 8–31% to 3–19% for Mistral-Small-3.1-24B. This shows that structural problems are addressed more reliably than semantic test errors, and that a single repair iteration, while beneficial on average, does not guarantee consistent improvements across all runs.

RQ3 (related to O3) *How well do the generated and repaired unit tests cover the code of the systems under test?* On the industrial repositories, Llama-3.3-70B achieves statement coverage of 34.1%–91.6% across modules, with branch coverage of 27.2%–70.5%, condition coverage of 29.4%–66.6%, and subroutine coverage of 75.7%–100.0%. On the CPAN distributions it reaches 65.5%–100.0% statement coverage. Mistral-Small-3.1-24B reaches considerably lower levels on both subject sets (statement 13.2%–63.8% industrial and 17.5%–100.0% on CPAN, with condition coverage dropping to zero in individual modules on both sets). Subroutine coverage is highest and statement coverage second for both models, while on the open-source side the order of the remaining two differs: branch coverage is higher for Llama-3.3-70B, condition coverage for Mistral-Small-3.1-24B. This indicates that generated tests invoke most functions but struggle to exercise conditional logic and edge cases. Llama also shows higher run-to-run consistency on the least stable module, Utils::ACL, where statement coverage varies with an SD of 8.8 compared to 19.5 for Mistral, confirming that model size affects not only coverage levels but also the reliability of test generation. One notable anomaly is DB2::DiffTables, where both models produce identical coverage, suggesting that neither generates tests exercising this module beyond a shared baseline.

The results on the three open-source systems show that the answers above are not specific to IBM's codebase. For Llama-3.3-70B, the pooled post-repair pass rate is 81.3% on the industrial systems and 80.5% on the open-source ones. For Mistral-Small-3.1-24B, the values are 74.2% and 57.6%, which shows that the model-size dependency is more pronounced outside the industrial setting.

For practice, the work shows that LLM-based test generation can already be used today to enhance productivity, especially for legacy systems with incomplete test coverage. The generated tests can serve as a starting point that developers review, adapt, and extend. This greatly reduces effort and enables faster establishment of tests. However, the approach in its current state is not suitable as a complete replacement for human test engineers. The insufficient branch and condition coverage, problems with automatic repair, and model-dependency of results still require human review and augmentation. For safety-critical systems or regulated areas, fully automatic test generation without review would not be sensible.

Future work should address several directions: developing error-class-specific prompt templates and repair strategies to reduce systematic failures such as `no_output` errors and repair degradation; investigating how far defensive execution should extend within a generated test file; performing prompt-pipeline ablations, including a single-phase baseline without business-logic extraction, to isolate the contribution of individual workflow phases; exploring adaptive prompt strategies, including meta-learning and prompt ensembles, to mitigate the observed model-dependency of results; evaluating prompt-development bias more directly by comparing model-neutral prompt templates, prompts optimized separately for each model, and prompts developed on one model and transferred to other models; assessing whether generated tests are accepted by project maintainers or contributors; con-

ducting long-term studies on the qualitative relevance, maintainability, and adoption of LLM-generated tests across development cycles; and improving trustworthiness through confidence scoring and coverage visualizations to support developer acceptance in practice.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s11219-026-09790-3.

Acknowledgements The authors thank Michael Simon (IBM) for his support in facilitating this project and Eray Gencay (IBM) for their valuable and constructive feedback throughout the project.

Author Contributions Simon Leunig: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Writing - Original Draft, Visualization. Kai Petersen: Supervision, Writing - Review & Editing.

Funding No funding was received for conducting this study.

Data Availability The supplementary materials include the publicly available prompt templates in Online Resource 1 and the Calculator example with its generation log in Online Resource 2. The four IBM source repositories analyzed in this study are proprietary and therefore cannot be shared publicly. No source code, generated tests, or per-function logs derived from those proprietary repositories are publicly available. The replication package for the three CPAN distributions, containing the prompts sent to the models, the generated and repaired tests, the per-stage test results, and the coverage measurements, is publicly available at https://github.com/simonleunig/perl-llm-test-generation.

Declarations

Competing interests The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Aleti, A. (2023). Software testing of generative ai systems: Challenges and opportunities. In *2023 ieee/acm international conference on software engineering: Future of software engineering (icse-fose)* (pp. 4–14).

Alshahwan, N., Chheda, J., Finogenova, A., Gokkaya, B., Harman, M., Harper, I.,... Wang, E. (2024). Automated unit test improvement using large language models at meta. *Companion proceedings of the 32nd acm international conference on the foundations of software engineering* (pp. 185–196). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3663529.3663839

Andrzejewski, M., Dubicka, N., Podolak, J., Kowal, M., & Siłka, J. (2025). Automated test generation using large language models. *Data*, *10*(10). https://doi.org/10.3390/data10100156. https://www.mdpi.com/2306-5729/10/10/156

Arora, C., Sayeed, A.I., Licorish, S., Wang, F., & Treude, C. (2024). Optimizing llms for code generation: Which hyperparameter settings yield the best results? *2024 31st Asia-Pacific software engineering conference (apsec)* (pp. 281–290).

Arteca, E., Harner, S., Pradel, M., & Tip, F. (2022). Nessie: automatically testing javascript apis with asynchronous callbacks. In *Proceedings of the 44th international conference on software engineering* (pp. 1494–1505). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3510003.3510106

Bajaj, Y., & Samal, M. (2023). Accelerating software quality: Unleashing the power of generative ai for automated test-case generation and bug identification. *International Journal for Research in Applied Science and Engineering Technology*, *11*, 345–350. https://doi.org/10.22214/ijraset.2023.54628

Bhatia, S., Gandhi, T., Kumar, D., & Jalote, P. (2024). Unit test generation using generative ai: A comparative performance analysis of autogeneration tools. In *Proceedings of the 1st international workshop on large language models for code* (pp. 54–61). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3643795.3648396

Braswell, W. (2024). The the perlgpt metacpan curator: Building a training dataset to fine-tune the perl llm. *The Science Perl Journal - Published in Austin, Texas, USA*, *1*, n. pag. https://doi.org/10.63971/spj.2024v01a03

Chen, Y., Hu, Z., Zhi, C., Han, J., Deng, S., & Yin, J. (2024). Chatunitest: A framework for llm-based test generation. *Companion proceedings of the 32nd acm international conference on the foundations of software engineering* (pp. 572–576). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3663529.3663801

Ghosh, T., & Abbas, N. 2026. Retry and refine: A multi-agent framework for llm based reliable unit test generation. In M. Bramer & F. Stahl (Eds.), *Artificial intelligence xlii* (pp. 92–105). Cham: Springer Nature Switzerland.

Go, G., Zhou, C., Zhang, Q., Jiang, Y., & Wei, Z. (2025). Lspai: An ide plugin for llm-powered multi-language unit test generation with language server protocol. In: *Proceedings of the 33rd acm international conference on the foundations of software engineering* (p. 144–149). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3696630.3728540

Granum, C. (2025). Test2::v0. https://metacpan.org/pod/Test2::V0. Accessed 05 Oct 2025

Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W.,... Liang, W. (2024). Deepseek-coder: When the large language model meets programming – the rise of code intelligence. arxiv:2401.14196

Guzu, A., Nicolae, G., Cucu, H., & Burileanu, C. (2025). Large language models for c test case generation: A comparative analysis. *Electronics*, *14*(11). https://doi.org/10.3390/electronics14112284. https://www.mdpi.com/2079-9292/14/11/2284

Haldar, S., & Capretz, L. F. (2026). Automated test plan generation using large language models. *Software Quality Journal,34*(1), 8. https://doi.org/10.1007/s11219-026-09744-9

Harman, M., Ritchey, J., Harper, I., Sengupta, S., Mao, K., Gulati, A.,... Robert, H. (2025). Mutation-guided llm-based test generation at meta. In *Proceedings of the 33rd acm international conference on the foundations of software engineering* (pp. 180–191). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3696630.3728544

IBM (2025) IBM watsonx.ai. https://www.ibm.com/products/watsonx-ai Accessed 05 Oct 2025

Jalil, S., Rafi, S., LaToza, T.D., Moran, K., & Lam, W. (2023). Chatgpt and software testing education: Promises & perils. In *2023 ieee international conference on software testing, verification and validation workshops (icstw)* (pp. 4130-4137).

Johnson, P. (2025). Devel::cover - code coverage metrics for perl. https://metacpan.org/pod/Devel::Cover. Accessed 05 Oct 2025

Johnson, P. (2026). Cpancover - coverage report. https://cpancover.com/. Accessed 26 Aug 2026

Karanjai, R., Hussain, A., Rabin, M. R. I., Xu, L., Shi, W., & Alipour, M. A. (2024). Harnessing the power of llms: Automating unit test generation for high-performance computing. arxiv:2407.05202

Kotti, Z., Dritsa, K., Spinellis, D., & Louridas, P. (2025). The fools are certain; the wise are doubtful: Exploring llm confidence in code completion. arxiv:2508.16131

Lenke, D., Ferrer, J., & Chicano, F. (2026). Influence of external dependency retrieval and prompt engineering in test case generation using llms. In L. Martínez et al. (Eds.), *Intelligent data engineering and automated learning – ideal 2025* (pp. 150–162). Cham: Springer Nature Switzerland.

Lops, A., Narducci, F., Ragone, A., Trizio, M., & Bartolini, C. (2025). A system for automated unit test generation using large language models and assessment of generated test suites. In *2025 Ieee International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (p. 29-36).

McLean, G. (2026). Xml::simple - an api for simple xml files. https://metacpan.org/release/GRANTM/XML-Simple-2.25. Version 2.25. Accessed 26 Aug 2026

Meta (2025). Llama 3.3 — model cards & prompt formats. https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_3/. Accessed 07 Oct 2025

Mistral (2025) Mistral small 3.1 — mistral ai. https://mistral.ai/news/mistral-small-3-1. Accessed 07 Oct 2025

Obaid Barraood, S., Mohd, H., Baharom, F., & Almogahed, A. (2023). Systematic literature review on test case quality characteristics and metrics. In *2023 3rd international conference on emerging smart technologies and applications (esmarta)* (pp. 01–08).

Paduraru, C., Stefanescu, A., & Jianu, A. (2024). Unit test generation using large language models for unity game development. In *Proceedings of the 1st acm international workshop on foundations of applied software engineering for games* (pp. 7–13). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3663532.3664466

Rahman, S., Kuhar, S., Cirisci, B., Garg, P., Wang, S., Ma, X.,... Ray, B. (2025). Utfix: Change aware unit test repairing using llm. *Proceedings of the ACM on Programming Languages*, *9*(OOPSLA1). https://doi.org/10.1145/3720419

Sarathy, G., & Ciornii, A. (2026). Tie::ixhash - ordered associative arrays for perl. https://metacpan.org/release/CHORNY/Tie-IxHash-1.23. Version 1.23. Accessed 26 Aug 2026

Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2024). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering,50*(1), 85–105. https://doi.org/10.1109/TSE.2023.3334955

Tang, Y., Liu, Z., Zhou, Z., & Luo, X. (2024). Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Transactions on Software Engineering,50*(6), 1340–1359. https://doi.org/10.1109/TSE.2024.3382365

The MetaCPAN Project (2026). Metacpan api. https://fastapi.metacpan.org/ (River statistics retrieved via the distribution endpoint, e.g., /v1/distribution/XML-Simple. Accessed 26 Aug 2026)

Walczak, J., Tomalak, P., & Laskowski, A. (2026). Impact of code context and prompting strategies on automated unit test generation with modern general-purpose large language models. *Journal of Systems and Software, 237*, 112834. https://doi.org/10.1016/j.jss.2026.112834. https://www.sciencedirect.com/science/article/pii/S0164121226000683

Wall, L., & Loukides, M. (2000). *Programming perl* (3rd ed.). USA: O'Reilly & Associates Inc.

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024a). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering,50*(4), 911–936. https://doi.org/10.1109/TSE.2024.3368208

Wang, Z., Liu, K., Li, G., & Jin, Z. (2024b). Hits: High-coverage llm-based unit test generation via method slicing. In *Proceedings of the 39th ieee/acm international conference on automated software engineering* (pp. 1258–1268). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3691620.3695501

Williams, K. (2026). Path::class - cross-platform path specification manipulation for perl. https://metacpan.org/release/KWILLIAMS/Path-Class-0.37. Version 0.37. Accessed 26 Aug 2026

Wong, K., Amayuelas, A., Pan, L., & Wang, W. Y. (2025). Investigating the transferability of code repair for low-resource programming languages. In L. Chiruzzo, A. Ritter, & L. Wang (Eds.), *Findings of the association for computational linguistics: Naacl 2025* (pp. 3410–3432). Albuquerque, New Mexico: Association for Computational Linguistics. https://aclanthology.org/2025.findings-naacl.190/

Yang, L., Yang, C., Gao, S., Wang, W., Wang, B., Zhu, Q.,... Chen, J. (2024). On the evaluation of large language models in unit test generation. In *Proceedings of the 39th ieee/acm international conference on automated software engineering* (pp. 1607–1619). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3691620.3695529

Yuan, Z., Liu, M., Ding, S., Wang, K., Chen, Y., Peng, X., & Lou, Y. (2024). Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering*, *1*(FSE). https://doi.org/10.1145/3660783

Zhang, Q., Fang, C., Gu, S., Shang, Y., Chen, Z., & Xiao, L. (2025). Large language models for unit testing: A systematic literature review. arxiv:2506.15227

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.