

Don't Bother to Use a Second LLM and Write Tests Yourself! A Study on Elixir

Noah Leu 

Lestio, Rostock, Germany

Julian Oertel  

Institute for Computer Science, University of Rostock, Germany

Regina Hebig¹  

Institute for Computer Science, University of Rostock, Germany

Abstract

Background. Generating software tests and implementations with Large Language Models (LLMs) is becoming more common. However, using the same LLM for the generation of implementation and tests might make both suffer from the same biases, potentially decreasing the tests' ability to detect faults compared to tests generated by a different LLM.

Aims. In this paper, we want to investigate this notion by combining different LLMs for generating implementation and tests.

Method. We generate Elixir code and test cases with 5 LLMs and compare them to a baseline of manually written code and tests. We wrote and generated 459 distinct implementations and 2709 test suites, manually analyzing 9417 test case failures.

Results. We find no difference between tests generated by the same or a different LLM as the code. However, of the 247 implementations flagged as faulty, 99 were only flagged by manually written tests.

Conclusions. Our results indicate that LLM-generated tests alone are not yet sufficient.

2012 ACM Subject Classification Software and its engineering → Software verification and validation; Software and its engineering → Functional languages; Software and its engineering → Source code generation

Keywords and phrases LLMs, Code Generation, Test Generation, Rare Languages

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.11

Category Technical Track Paper

Supplementary Material *Collection (Source Code and Results):* <https://doi.org/10.5281/zenodo.20157738> [13]

1 Introduction

Software testing is critical in ensuring software systems' quality, reliability, and maintainability. Manually written unit tests, integration tests, and end-to-end tests have proven effective but remain costly and time-intensive [31]. As development often happens under time and budget constraints, there is great interest in generating tests automatically addressed by tools like EvoSuite [9] and Randoop [22]. In recent years, Large Language Models (LLMs) have made it possible to approach generating tests not just from code, but directly from textual specifications [30, 34]. As studies show promising results for LLM-assisted test generation [2, 8], LLMs are increasingly integrated into test generation environments, such as TestSpark [26].

¹ Corresponding author



© Noah Leu, Julian Oertel, and Regina Hebig;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;

Article No. 11; pp. 11:1–11:20



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Simultaneously, there is increasing interest in generating the code itself from the specifications [15, 20]. This leads to the following potential problem: When looking at human developers, we accept the idea that different developers might have different biases and thus there is value in involving multiple individuals during coding and quality assurance work. For example, code reviews or pair programming are two widely accepted methods in industry [28, 18]. However, when the same model generates both the implementation and its tests, there is a risk that incorrect assumptions and biases present in the initial code generation are reinforced and introduced to the test generation as well. Thus, an LLM may be less likely to generate tests that detect errors it made itself during implementation generation than errors another LLM made during implementation generation. This leads to the following research question:

RQ: Are tests better at detecting faults when code and tests are not generated by the same LLM?

We perform an experiment with 9 specifications derived from a start-up's software system and 5 LLMs. Preceding this, we conduct a pre-study to develop and refine the prompts and specification format and select suitable LLMs in order to minimize compilation errors and facilitate the generation of effective tests. Including manually written tests and code, we work with 459 distinct implementations and 2709 test suites. We manually analyze 9417 test case failures to identify their root causes and distinguish whether faults in the implementation or faults in the test caused the failures.

Most studies on LLM-based test generation focus on widely adopted languages such as Java, Python, or JavaScript [4, 21, 36, 27]. These ecosystems benefit from large amounts of publicly available training data. In contrast, LLMs tend to perform worse when generating for low-resource languages [5, 12, 11]. As this study was performed in collaboration with a start-up and the language infrastructure they use, we focus on such a language: Elixir and its Phoenix web framework. Despite representing a smaller share of total practitioners, Elixir constitutes a meaningful ecosystem with steady industry relevance, as evidenced by a recent developer survey².

To this end, we make the following contributions: (1) Insights into the question whether using the same LLM for code and test generation is of disadvantage. (2) Insights into the applicability of LLM code and test generation for low-resource languages like Elixir. (3) An extensive dataset of LLM-generated code and test cases for Elixir, including baseline manual implementations and tests.

This paper is structured as follows. We discuss related work in Section 2. Then we present our pre-study in Section 3 and the methodology in Section 4. Results are shown in Section 5 and discussed in Section 6. We conclude in Section 7.

2 Related Work

Numerous studies examine the potential to use LLMs to automate or assist in test creation across a range of programming languages and domains [34]. Some recent studies point out that execution, compilation, and logical validity can still be inconsistent in LLM-generated test suites [36, 37]. For example, tests generated using ChatGPT (GPT-3.5-turbo) in the study of Yuan et al. compiled only in 42.1% of cases [36]. Bhatia et al. [4] investigated LLM-generated unit tests in Python using ChatGPT, comparing them against traditional tools like Pynguin [16]. While the coverage was often comparable, nearly a third of the generated

² see <https://survey.stackoverflow.co/2025/technology/>

assertions were incorrect. Nashid et al. [19] showed that examples in the prompts can increase the accuracy of the generated assertions to up to 76%. Examples like ChatUniTest demonstrate promising results compared to traditional tools for test generation such as EvoSuite or TestSpark [6]. Despite these advancements, multiple studies note that LLM-generated tests often underperform when compared to tools like EvoSuite [9], especially in detecting subtle or complex bugs [21, 1, 29].

Several researchers suggest improvements to how LLMs can be used for test generation. For example, Yang et al. explore the optimization of prompt clarity and structure for test generation [35]. Prompts specifically adapted to an LLM, stripped of unnecessary context, and augmented with few-shot examples have shown increased effectiveness in code generation [35, 25]. Ouedraogo et al. and Ryan et al. decompose the generation process into smaller reasoning steps to improve correctness and test quality, especially for complex logic or branching scenarios [21, 24]. Further Ryan et al. and Pizzorno et al. integrate traditional coverage feedback mechanisms into the test generation loop. This allowed models to iteratively improve their generated tests by being motivated to explore untested paths [3, 24]. Li et al. [14] focus on bug detection capabilities of LLMs and propose a follow-up question method. Chowdhury et al. employed static program analysis to enhance LLM-based test generation for Java [23]. This enabled better targeting of edge cases and execution paths.

Niche or low-resource languages such as Elixir pose unique challenges to LLMs due to their limited presence in training data and unique abstractions [5, 12, 11]. Few studies apply LLMs on such languages. One example for this is the paper by Tarassow et al. that employed GPT-3.5 to better understand and generate documentation or implementations for low-resource or domain-specific languages [30]. This study also explored ways to assess implementations or tests generated by LLMs, which is not a trivial task.

3 Pre-study

To enable the candidate LLMs to generate valid code and test suites, we conducted a pre-study targeting the general structure and degree of detail and verbosity of specifications and the prompts used. The pre-study started with initial versions of the prompts that were refined in several iterations. We stopped iterating when the prompts led to implementations that were mostly free of compilation errors when applied to the candidate LLMs for two trial specifications. The first trial specification requests a minimal LiveView rendering a `<h1>` element, and the second one aims at computing the minimum price from a list of products.³

3.1 Candidate LLMs

We considered different closed-source and open-weight models, specifically, the candidates GPT-4.1, Claude Sonnet 3.7, and Gemini 2.5 Pro (Google DeepMind) for their strong results in code-related benchmarks such as LMArena⁴, GPT-4o-mini as it is a smaller model that is optimized toward speed and efficiency, and Llama 4 Maverick and DeepSeek V3 as they were trained under different resource constraints and data diversity (our study was performed in April/May 2025).

Based on the results of the pre-study, we excluded Gemini due to its unreliable output formatting – which prevented automated parsing – and frequent token limit violations that led to incomplete code files. Prompt adaptations failed to mitigate these behaviors.

³ The trial specifications can be found in the supplemental materials (cf. Section 7)

⁴ <https://lmarena.ai/leaderboard/webdev>

3.2 Prompts

We developed prompts for the generation of code as well as test suites. The initial prompts were inspired by current practices in prompt engineering [35, 17, 25]. Specifically, the initial drafts were adapted from publicly available resources on the community-driven platform Cursor Directory⁵ and other sources such as GitHub⁶. On Cursor Directory, users can publish system prompts to be used with the IDE called Cursor for many languages and frameworks. The prompt published under “Elixir Cursor Phoenix Rules”⁷ was identified as the foundation, as it outlined essential conventions and best practices for generating Elixir code within Phoenix applications. However, the initial prompts led to problems, such as missing required imports, unspecified dependencies, or the use of non-existent functions. Our goal was to minimize the generation of non-compiling implementations, as these are not usable to compare the quality of generated tests. In addition, we aim to give the LLMs the best possible chance to generate good test suites for answering our research question.

We made the following changes to the prompts⁸: **(1)** We instructed the LLMs to avoid unspecified external libraries. **(2)** For recurring issues, we added specific usage examples. These included variations with detailed type annotations, example-based few-shot inputs, and minimal instruction variants. For example, mocking external function calls posed a special challenge. Trial runs of the pre-study resulted in inconsistent model behavior when asked to use third-party libraries for mocking dependencies. These libraries often required configuration setups in multiple files for each dependency used. To address this issue, we decided to use the Mock⁹ library, which allows mocking using only the file of the test suite itself. A single-shot example demonstrating correct mocking syntax was included in the testing system prompt. **(3)** To force the LLM to adhere closely to the specification we added a sentence explicitly asking the LLM to do so. **(4)** Due to issues with LLMs explaining their outputs, we also include the instruction to only return code snippets that work as standalone modules and can be directly used as an Elixir file.

For test generation we explored both white-box (using the implementation code as context) and black-box (using only the specification) test generation. However, the white-box approach resulted in better outcomes, such as finding implementation faults like incorrect data type handling and value edge cases. This finding is also supported by a study by Guzu et al. in 2025 [10]. We therefore decided on white-box test generation.

3.3 Specification Format

The initial specification format used a simple textual description. This resulted in rather poor performance for some cases where the LLM did not understand or correctly perform the initial tasks. We managed to improve the generation results by adding examples, including input and output pairs, and using a more structured format with bullet points, reducing the number of filler words.

The final specification format uses concise context information, e.g., module name, input/output expectations, functional description, as well as necessary module imports and aliases. This was also necessary to ensure that human-written black-box tests could be used on the generated implementations. The schema with headings, subheadings, and bullet points included:

⁵ <https://cursor.directory/plugins/elixir>

⁶ <https://gist.github.com/kostysh/dbd1dfb2181b96563754222903bf67e7>

⁷ <https://cursor.directory/plugins/elixir>

⁸ The resulting prompts are in the supplemental materials (cf. Section 7).

⁹ <https://hexdocs.pm/mock/Mock.html>

- **Module and function name:** Specific identifiers to ensure uniform access across tests
- **Purpose:** A high-level description of the intended functionality
- **Inputs and outputs:** Detailed argument and return type information
- **Constraints:** Requirements such as avoiding external libraries
- **Known edge cases and examples:** Representative scenarios for correctness
- **Notes:** Optional attribute for additional guidance for realistic implementation behavior

4 Methodology

We employ a controlled experimental design to evaluate LLM-based test generation in a realistic context with requirement specifications from a real startup. The startup provided industry-relevant tasks in the form of functional requirements but had no further involvement.

Table 1 defines the central terms used throughout the paper.

■ **Table 1** Definitions of the study’s key terms and metrics.

Term	Definition
Test case failure	An individual test case that does not pass, or a test file that fails to compile. Raw outcome; not yet attributed to a cause.
Implementation error	A test case failure whose manually determined root cause lies in the code (violates the specification).
Test error	A test case failure whose root cause lies in the test (e.g., wrong assertion, test compile error).
Faulty implementation	An implementation with ≥ 1 failure attributed to an implementation error (can contain several).

4.1 Task Specifications

4.1.1 Source

The task specifications used in this experiment were derived from an actual software product provided by the collaborating startup. The underlying architecture of the system includes a web application that handles user interactions and recommendation logic. The system incorporates multiple functional components, such as scoring and ranking algorithms, and recommendation filtering based on user input and product attributes. These components offer a diverse set of functions, ranging from simple utility operations to more complex logic involving condition-based decisions and weighted scoring. Each specification used in the experiment reflects a requirement within the platform and was documented textually.

4.1.2 Complexity Levels

To evaluate the influence of task difficulty on LLM performance, the nine specifications were grouped into three complexity categories (see Table 2). We chose a distribution of four simple, three medium, and two complex tasks. A higher number of simple tasks increases the likelihood of obtaining sufficient compiling implementations while keeping the manual error classification workload manageable.

■ **Table 2** Specification complexity levels and characteristics. (LOC $\equiv$ Lines of Code).

Level	Key Characteristics	Expected LOC	Count
Simple	Minimal logic, branching, and data gathering	≤ 30	4
Medium	LiveView, complex outputs (maps, changesets)	≤ 90	3
Complex	Process-level (GenServer) and dependencies	> 100	2

■ **Table 3** Overview of the nine function specifications used in the experiment (* = Complexity).

*	Name (ID)	Description
Simple	1. Get All Brands	Returns a case-normalized list of unique brand names.
	2. Get Average Price	Calculates the average product price, ignoring nil values.
	3. Get Minimum Attribute	Retrieves the minimum value of an attribute across a product list.
	4. Get Reading Duration	Estimates reading time (in minutes) for a given blog post text.
Medium	5. Validate Contact Form	Validates form input fields and returns success or error details.
	6. Validate Password	Validates password complexity in an Ecto changeset.
	7. Product List LiveView	Renders a product list dynamically using Phoenix LiveView.
Complex	8. Slugify	Converts text to URL-friendly slugs with optional customization.
	9. Product Cache GenServer	Implements a GenServer-based product cache using ETS for fast retrieval.

4.1.3 Specification Selection

To simplify the evaluation and maintain comparability, all functions were designed to be implemented within a single Elixir module. This approach reduces environmental complexity and ensures that generated test suites can be executed using automation scripts. Specifications that required multiple modules or intricate application contexts, such as full Phoenix LiveView setups that include router configuration, were excluded from this study. Where applicable, each specification also aimed to contain only a single entry function to be tested. This ensures that test suites are provided with a specific entry point to the implementation. Additionally, all functionality with database requirements relies on mocked dependencies rather than direct database access. Features requiring integration-level setup, such as sandbox databases, were excluded, as these would again require multiple file adaptations.

The specifications cover a range of tasks from simple utility functions to full process-level modules. Importantly, the nine specifications are different from the two trial specifications used in the pre-study. This was crucial to prevent the developed prompts and specification structure from being specific to the specifications used in this study. Table 3 summarizes the specifications grouped by complexity.¹⁰

4.2 Large Language Model Interaction and Settings

We decided to use the LLMs **GPT-4.1**, **GPT-4o-mini**, **Claude Sonnet 3.7**, **Llama 4 Maverick**, and **DeepSeek V3**. To access the models, we used OpenRouter to provide a consistent API across multiple providers which simplified logging of prompts, responses, and metadata as well as addressing rate limits.

¹⁰The complete specifications, including input-output formats, constraints, edge cases, and example calls, are provided in the supplemental materials (cf. Section 7).

■ **Table 4** Default generation settings applied to all LLMs.

Parameter	Value
Temperature	0.0 (deterministic, greedy decoding)
Top-p	1.0 (full probability distribution)
Top-k	1 (always choose the most probable token)
Token limit	3000 tokens (implementations); 6000 tokens (test suites)

All generations were performed using a consistent set of decoding parameters and resource constraints, which were applied to all models during implementation and test suite generation (see Table 4). By setting the temperature to 0.0 and restricting Top-k to 1, generation was made as deterministic as possible, as we aim to observe structural differences in the models’ reasoning and minimize randomness in sampling. Implementations were limited to 3000 tokens and test suites to 6000 tokens (to accommodate several detailed test cases and setup logic). These limits respect the expected file sizes per complexity level while bounding output size, and curb the infinite code repetition observed in the pre-study. When a file was truncated because of repetition and failed to compile, we classified it as a syntax error. To avoid unintended bias from prior interactions, every request was executed in a new session, ensuring that previous generations do not directly influence later runs.

4.3 Running the Experiment

4.3.1 Implementation and Test Generation and Execution Workflow

The workflow consisted of two main stages: manual solution and LLM-based solution. Before the generation with LLMs began (but after the pre-study), the first author implemented solutions to the nine task specifications as well as corresponding black-box test suites manually. Using black-box tests instead of white-box tests made a manually written baseline possible, as it was sufficient to write one test suite per specification instead of one for each of the up to 459 expected implementations.

In the second stage, the LLM-based solutions and tests were created in 10 runs to account for variability in LLM outputs. Each run involved all nine specifications and included three steps. (1) every LLM generated one complete Elixir implementation for each specification. (2) Each LLM generated a test suite for each of the implementations (including the human-written implementation). (3) after successfully generating all files, the execution followed an automated test execution procedure. For each implementation, the implementation file was placed in the corresponding directory of the Phoenix project and compiled. If the compilation failed, the implementation was marked as invalid, and no test suite executions were attempted for this specification. In this case, the test suites were neither executed nor compiled. When compilation succeeded, the process continued with executing each test suite for this implementation.

4.3.2 Dataset Size

The total dataset is derived from 10 runs, 9 specifications, and 5 models. As detailed in Table 5, we produced 459 implementations and 2,709 test suites. To ensure comparability, manual implementations were treated as unique for each of the 10 runs, allowing the same number of test suites to be generated for manual code as for any LLM/specification combination.

While the theoretical maximum for test executions is 3,240 (Table 5), the actual number of executed tests is lower whenever an implementation fails to compile. Each compilation failure results in the loss of six potential executions: five LLM-generated test suites and one human-written test suite.

■ **Table 5** Overview of expected set of created implementations (I), expected set of created test suites (T), and maximum number of test suite executions (E) across 10 runs (R) for each of the 9 specifications (S) and each of the 5 models (M).

	Per Specification	All Specifications
I_M (manually created implementations)	1	$(I_{M_{spec}} \times S) = 9$
I_G (generated implementations)	$(R \times M) = 50$	$(I_{G_{spec}} \times S) = 450$
Sum $I = I_M + I_G$	51	459
T_M (manually created test suites)	1	$(T_{M_{spec}} \times S) = 9$
T_{GG} (generated test suites for I_G)	$(I_{G_{spec}} \times M) = 250$	$(T_{GG_{spec}} \times S) = 2250$
T_{GM} (generated test suites for I_M)	$(R \times M) = 50$	$(T_{GM_{spec}} \times S) = 450$
Sum $T = T_M + T_{GG} + T_{GM}$	301	2709
E_{MM} (executions of T_M on I_M)	$R \times T_{M_{spec}} = 10$	$R \times T_{M_{all}} = 90$
E_{MG} (executions of T_M on I_G)	$I_{G_{spec}} = 50$	$I_{G_{all}} = 450$
E_G (executions of T_{GG} and T_{GM})	$(T_{GG_{spec}} + T_{GM_{spec}}) =$	$(T_{GG_{all}} + T_{GM_{all}}) =$
	300	2700
Sum $E = E_{MM} + E_{MG} + E_G$	360	3240

4.4 Analysis

In this subsection, we describe our analysis procedure.

4.4.1 Root Cause Examination

The output of the test suite was parsed to classify the results into: **Test compilation fails**, i.e., test files failed to compile. **All tests pass**, i.e., no failures are detected during execution of the tests, indicating no defects. **Tests fail**, i.e., failures were detected during test execution.

All failures were further classified regarding their root cause. To gain a first understanding of error types present, the output for all failed test cases or test compilations was closely examined to find repeating patterns. This resulted in an initial categorization of errors into types such as `assertion_failure`, `undefined_function`, or `regex_error`. However, these error types were not enough to clearly determine whether a failure was caused by an issue in the implementation or in the generated test suite. To accurately answer that question, a manual root cause analysis was performed. This process involved inspecting the specification file, the generated implementation file, the corresponding test file, and the error output for every recorded failure across all runs.

While some error types can be confidently attributed to one side (e.g., syntax errors in test files indicate test failures), many cases require precise examination. For example, when a test failed due to not receiving expected function calls, the analysis needed to verify whether the implementation did not make the expected calls or the test awaited an incorrect function call. A large spreadsheet was created to summarize each error type, affected specifications, number of occurrences, counts of implementation and test errors, and lastly include the manually identified root causes. All in all, 9417 test case failures were investigated manually (there can be multiple test failures for one test suite execution).

4.4.2 Statistical Evaluation

The goal was to determine whether observed differences between models and test generation strategies are statistically significant. Since the data did not follow a normal distribution and included ordinal characteristics (e.g., number of detected defects, failure rates), non-parametric tests were selected.

We formulated the following null hypotheses for our research question:

H1₀: There is no difference in test error counts between the LLMs used to generate test suites.

H2₀: There is no difference in the number of detected implementation errors between test suites generated by the same model as the code, test suites generated by a different model as the code, and test suites written by a human.

In addition, we tested following supporting null hypotheses to better understand the data:

H3₀: There is no difference in the number of detected implementation errors between LLMs used to generate the implementations.

H4₀: There is no difference in the number of implementation errors across complexity levels.

H5₀: There is no difference in the number of test errors across complexity levels.

H6₀: There is no correlation between the code coverage of a test and the number of detected implementation errors.

We used Kruskal-Wallis as an omnibus test to see whether at least one model performs differently. Allowing the detection of which exact pairs differ, a pairwise Holm-corrected Mann-Whitney U test was used, when Kruskal-Wallis indicated significant differences. We apply a significance level of $\alpha = 0.05$ for all tests. For testing **H6₀**, we use Spearman's ρ due to the count-like, skewed nature of errors. Statistical significance and 95% confidence intervals were estimated via bootstrapping (1000 iterations). Effect sizes were calculated for **H1₀ - H5₀** using common language effect size for Mann-Whitney U tests and η^2 for Kruskal-Wallis tests. The effect sizes were classified into four levels (negligible, small, medium, large) following [32] and [7, 33], respectively.

5 Results

We present compilation rates, the error type classification, and results of statistical tests.

5.1 Compilation Rate

Implementation compilation rate. In total, 24 implementations did not compile. Table 6 summarizes the compilation success rates. For simple specifications, almost all models achieved near-perfect compilation success. Medium-level tasks showed minor reductions in compilation success for GPT-4.1 and GPT-4o-mini. The most noticeable decline occurred in complex specifications, where GPT-4.1 and GPT-4o-mini dropped to 60% and 65%, respectively. Claude Sonnet 3.7 achieved a 100% compilation rate across all complexity levels. Compilation failures were often related to language-specific constraints rather than a complete misunderstanding of the task. Especially problems within guard clauses and incorrect imports suggest that the LLMs occasionally failed to follow Elixir's rules regarding language-specific functionality usage. Issues regarding remote function calling and guard-related errors point to challenges unique to functional programming.

Test suite compilation rate. Test compilation success was measured under the condition that the corresponding implementation compiled successfully. Thus, test suites for which the implementations do not compile are not considered. For simple specifications, most models achieved very high compilation success rates, with Claude Sonnet 3.7 reaching 100%. Llama 4 Maverick showed a slightly lower rate at 92.89%, indicating some issues even in basic tasks. GPT-4o-mini, however, achieved a lower rate of 83.19%. Medium complexity specifications led to a significant decline for GPT-4.1 and Llama, with success rates dropping to 61.02% and 56.82%, respectively. In complex scenarios, the trend remained similar, with GPT-4.1

■ **Table 6** Implementation and test compilation success rate by source of implementation/test and complexity (successfully compiled implementations/created implementations (rate) and successfully compiled test suites/considered test suites (rate)).

Model	Implementation compilation			Test suite compilation		
	Simple	Medium	Complex	Simple	Medium	Complex
GPT-4.1	40/40 (100%)	28/30 (93.33%)	12/20 (60%)	235/239 (98.33%)	108/177 (61.02%)	60/100 (60%)
GPT-4o-mini	40/40 (100%)	29/30 (96.67%)	13/20 (65%)	198/238 (83.19%)	150/177 (84.75%)	100/100 (100%)
Claude	40/40 (100%)	30/30 (100%)	20/20 (100%)	239/239 (100%)	161/177 (90.96%)	83/99 (83.84%)
Sonnet 3.7	39/40 (97.5%)	30/30 (100%)	18/20 (90%)	222/239 (92.89%)	100/176 (56.82%)	86/100 (86%)
Llama 4	40/40 (100%)	30/30 (100%)	17/20 (85%)	234/238 (98.32%)	162/177 (91.53%)	77/99 (77.78%)
Maverick						
DeepSeek V3						
Manual	4/4 (100%)	3/3 (100%)	2/2 (100%)	4/4 (100%)	3/3 (100%)	2/2 (100%)

continuing to perform poorly at 60% success, while Claude stayed above 83%. Interestingly, GPT-4o-mini compiled slightly more test suites successfully on medium specifications than on simple ones and even reached a 100% compilation rate for complex specifications.

5.2 Resulting Test Execution Dataset

As discussed in Subsubsection 4.3.2, the actual number of test suite executions can be below the maximum of 3240 possible test suite executions. In practice, 24 implementations failed to compile, causing the lack of 144 test suite executions. 20 of those failures were attributed to Claude Sonnet 3.7 for complex specifications. In addition, an unknown error occurred with the generation of the test suites, resulting in five cases in the API returning either empty text or an error. These five cases could not be assigned to failure of the LLMs. They may be the result of a timeout on the server side of the API provider, OpenRouter. Therefore, these 5 cases were considered invalid and not considered for further study. Within the 3091 test suite execution attempts, 360 test suites were not able to compile. A total of 28542 single test cases were executed. From these cases, 19125 (67%) cases passed without failing. 9417 (33%) single test cases failed during execution. Of those, 5817 (62%) failed due to test errors and 3600 (38%) failed due to implementation errors, i.e., they successfully detected errors. Note that single test cases can fail due to the same implementation fault.

5.3 Error Type Classification

Table 7 summarizes the identified error types and their occurrences. This also includes syntax errors in the test cases that failed to compile. While there are error causes that are specific to implementations or tests only, several error categories include errors that can occur both in implementations and in tests. Most prominently, this holds for `assertion` failures, which are caused by errors in tests as well as errors in the implementations. A table with detailed error types is available in the supplemental material (cf. Section 7)

Test and Assertion Failures. These errors occur when code executes but does not behave as the tests expect. This category includes classical assertion failures (3,072), structural mismatches (717), and cases where expected exceptions are not raised (65) or telemetry

■ **Table 7** Summary of identified error categories and their distribution. Total errors are split into test errors and implementation errors (test + impl.). We also considered non-compiling test suites in this without breaking them down into individual test cases.

Category	Total Errors	Primary Cause
Test & Assertion	4233 (1650+2583)	Incorrect assertions/expectations
Runtime & Language	3716 (2733+983)	Invalid assumptions/API misuse
Erlang & Process	1648 (1614+34)	Process lifecycle/concurrency
Compilation	165 (165+0)	Syntax/Language constraints
Test Structure	15 (15+0)	Misuse of testing frameworks
Total	9777 (6177+3600)	

messages are not received (348). Additionally, doctests (embedded code execution examples in the implementation) fail (30). This shows that LLMs sometimes generate incorrect or inconsistent code samples directly in their implementation or do not correctly cover their own examples functionally.

Runtime and Language Errors. These occur during execution, typically due to invalid assumptions regarding values, functions, or data structures. Common examples include `undefined_function` (1,107) arising from spelling errors or incorrect API assumptions, and `argument_error` (813) caused by improper parameter usage. Scoping issues are reflected in `undefined_variable` (10), while `function_clause` errors (838) often stem from incorrect pattern matching or insufficient guard clauses.

Erlang, Process and Concurrency Errors. These involve the underlying Erlang VM and process management, particularly within the `cache_genserver` specification. Errors often arise from improper process handling, such as `process_already_started` (258) due to failed test isolation or `named_table_exists` (224) during ETS table creation. Additionally, attempts to mock Erlang-level modules led to 524 `sticky_module` errors, preventing module reloading. Remaining errors included `process_errors` (1) that stem from communication or termination failures and `erlang_errors` (17) due to malformed terms or invalid Erlang API usage.

Compilation and General Language Errors. These are identified by the compiler before execution and stem from invalid syntax or unsupported constructs. Notably, errors like `pin_operator_error` (55) and `invalid_use_of_underscore` (49) suggest that LLMs occasionally adopt syntax habits from other programming languages.

While more of the error found were rooted in incorrect tests than in implementations, it cannot be concluded that LLMs are worse at generating tests, since we generated more test suites (2700) than implementations (450).

Observations: The most frequent error by a wide margin is `assertion_failure`, accounting for 3072 total cases (with 2267 implementation errors). Implementation errors are most commonly detected as `assertion_failure` (with 2267 errors), but also as `match_failures` (195), `argument_errors` (225), `undefined_functions` (151), `bad_map_errors` (248), and `protocol_errors` (147). Erlang, Process and Concurrency Errors are almost exclusively caused by faulty tests.

11:12 Don't Bother to Use a Second LLM and Write Tests Yourself! A Study on Elixir

■ **Table 8** Mean code coverage, in terms of share of lines of code covered in a run of the test suite, and number of identified faulty implementations by test suite source.

Source of Test Suite	Mean code coverage in %			Faulty implementations found	
	Simple	Medium	Complex	By	Only by
GPT-4.1	94.09	59.91	35.01	55	2
GPT-4o-mini	77.98	69.97	73.62	89	3
Claude Sonnet 3.7	95.76	83.40	61.73	94	4
Llama 4 Maverick	88.60	36.49	65.66	76	20
DeepSeek V3	92.16	80.22	60.43	94	5
Manual	95.41	94.13	89.43	222	99

■ **Table 9** Statistical comparison of test error counts by source of test suites (LLMs or manual). Only significant results of the pair-wise comparison are shown.

Comparison	H-statistic	p-value	η^2	Significant
Kruskal-Wallis (by LLM)	28.493	< 0.0001	0.010 (negligible)	Yes

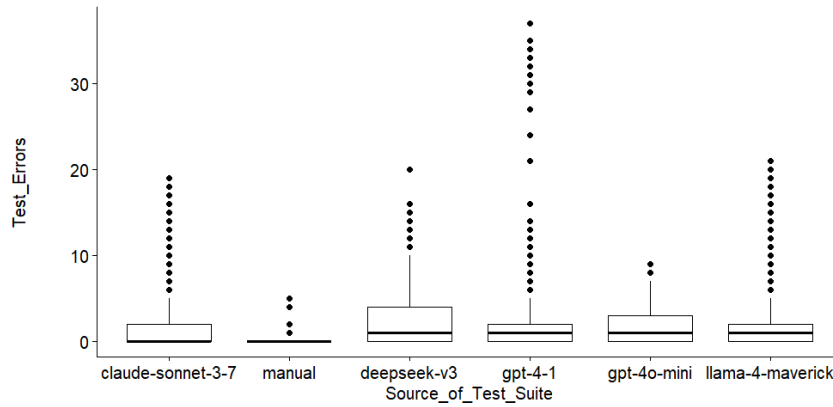
Comparison	U-statistic	p-value	CLES	Significant
Manual vs LLMs	437815.0	< 0.0001	0.67 (medium)	Yes

Comparison	U-statistic	p-value	CLES	Significant
Claude vs DeepSeek	117462.5	0.0057	0.556 (negligible)	Yes
Claude vs GPT-4.1	112876.5	0.0001	0.575 (small)	Yes
Claude vs GPT-4o-mini	111098.5	< 0.0001	0.581 (small)	Yes

5.4 Code Coverage and Identified Faulty Implementations

Table 8 presents the mean code line coverage per test generation model by specification complexity level. For simple specifications, most models achieved high coverage values above 88%. Claude Sonnet 3.7 and DeepSeek V3 are among the top-performing models for medium and complex tasks, while Llama 4 Maverick showed strong results for simple tasks but suffered from the lowest medium-level coverage (36.49%). This is due to the mean coverage of 13.33% and 10.14% for the specifications `product_liveview` and `validate_password`, respectively. Interestingly, the manually written black-box test suites achieved the highest coverage nearly across all levels (being second highest for simple specifications). Table 8 also shows the number of faulty implementations flagged by the tests created by each LLM. An implementation can include multiple faults. Here, we only count those faulty implementations for which at least one fault was identified correctly by at least one test suite. Altogether, 247 implementations were flagged as faulty (5 of the 9 manual implementations and 242 of the 450 generated implementations). Most faulty implementations were flagged by the manually written test suites. The table also shows the number of faulty implementations that were only identified by test suites from the given source, and no other test suites. E.g., for 20 faulty implementations, only test suites generated by Llama identified errors, but no other test suites.

Observations: Code coverage tends to be higher for manually created black-box tests than for generated white-box tests. Most faulty implementations identified by test suites from a single source, only, were identified by manually written test suites.



■ **Figure 1** Test errors per source of test suite (LLM or manual).

5.5 Statistical Evaluation

The tested dataset contains 3091 entries – one entry per test suite execution attempt.

5.5.1 Test Error Counts by Test LLM

Table 9 shows an overview of the tests regarding hypothesis $H1_0$ that there are no differences in the number of test errors based on the LLM that created a test suite. A Kruskal-Wallis test confirmed statistically significant differences in test error counts among the source of the generated test suites ($H = 28.493$, $p < 0.0001$), rejecting hypothesis $H1_0$. Pairwise comparisons show that Claude Sonnet 3.7 differed significantly with small effect size compared to several other models. As shown in Figure 1, Claude introduced slightly fewer test errors (mean: 2.07, median: 0) compared to GPT-4.1 (mean: 2.61, median: 1) and DeepSeek V3 (mean: 2.48, median: 1). The comparison to GPT-4o-mini (mean: 1.96, median: 1), however, is not quite as obvious, as Claude has a worse average due to outliers, but overall a better median. The comparison between the manually written test suites (mean: 0.49, median: 0) and all LLM-generated suites was significant, with a medium effect size. The manually created tests have a lower average error count.

Observation: While $H1_0$ is to be rejected, the differences between the LLMs are often of small effect sizes or even negligible. For manually written tests, fewer test errors were observed than for the generated tests.

5.5.2 Implementation Error Counts by Source of Test Suite

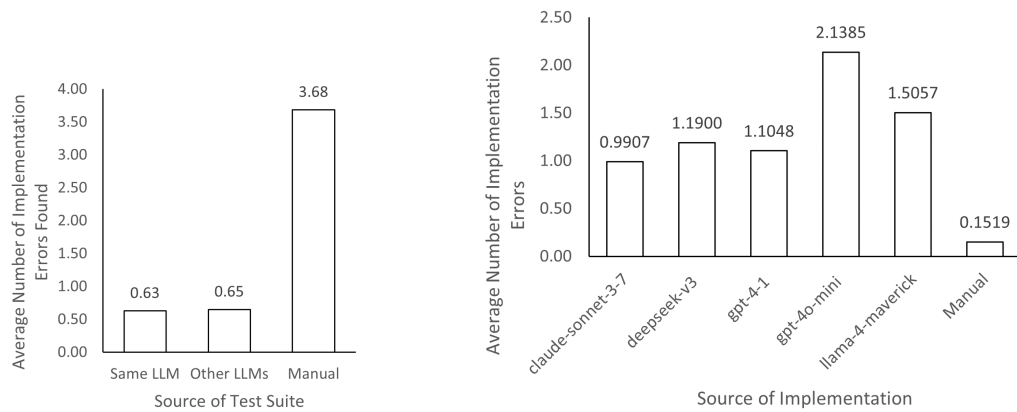
Table 10 summarizes the tests of hypothesis $H2_0$. It shows that the difference in the source of the tests compared to the source of the implementation impacts the number of implementation errors that can be found. We reject $H2_0$ as the Kruskal-Wallis test showed a statistically significant difference between the three testing scenarios, suggesting that the source of the test suite does impact the implementation error counts found.

Pairwise comparisons revealed that a significant relationship exists between the implementation faults found by the manually created test suites and faults found by the same model (“Same”) or other models (“Other”). In both cases, we found a medium effect size with manually written tests finding many more implementation errors on average than

■ **Table 10** Statistical comparison of implementation error counts by source of test suite (manual, same LLM, other LLM).

Comparison	H-statistic	p-value	η^2	Significant
Kruskal-Wallis	223.499	< 0.0001	0.070 (medium)	Yes

Comparison	U-statistic	p-value	CLES	Significant
Manual vs Same	173767.0	< 0.0001	0.643 (medium)	Yes
Manual vs Other	712926.0	< 0.0001	0.653 (medium)	Yes
Same vs Other	540217.0	0.1520	0.513 (negligible)	No



(a) By test suite source.

(b) By implementation source.

■ **Figure 2** Comparison of average implementation errors found based on different sources.

LLM-generated tests (see Figure 2a). However, there was no significant difference between the implementation errors found by test suites from the same LLM that generated the implementations vs. other LLMs. Thus, changing the LLM when generating tests for generated code might not increase the chances of finding errors.

Observation: $H2_0$ can be rejected as manually created tests find more implementation errors than generated tests. However, there is no significant difference between the number of implementation errors found by tests generated from the same vs. other LLMs than the LLM that generated the implementation. Thus, the answer to the **RQ** is: **No**, tests are not better at detecting faults when tests and code are not generated by the same LLM.

5.5.3 Additional Tests¹¹

Testing hypothesis $H3_0$ showed that there is a significant difference between the implementation errors produced by most pairs of LLMs (with the exception of Claude vs. GPT-4.1 and DeepSeek vs. Llama). However, effect sizes for these differences are small to negligible. Similarly, we found a significant difference with small effect size between the number of implementation errors in generated and manual implementations (see Figure 2b).

¹¹Detailed test results for hypothesis tests $H3_0$ to $H5_0$ can be found in the supplemental materials (cf. Section 7).

Testing hypotheses $H4_0$ and $H5_0$ showed significant differences in the number of implementation errors and test errors that can be found in tasks with different complexity levels. The only exception is the difference between the number of implementation errors in simple and medium tasks, which was not significant in our test. Effect sizes for implementation errors were small. For test errors, the effect sizes were medium to large.

Finally, we used Spearman's ρ to test $H6_0$. We looked at data from faulty implementations, i.e., implementations for which at least one test found an implementation error. There was a significant but weak positive correlation between the code coverage and implementation errors found by a test ($df = 1611$, $95\% CI = [0.062, 0.154]$) with $\rho = 0.109$.

Observations: The model used for code generation influences the number of implementation errors. Complexity levels influence the number of test and implementation errors. We find a weak correlation between code coverage of tests and the number of found errors.

6 Discussion

In the following, we discuss implications of our findings, compare them to related work, and present threats to validity.

6.1 Implications

The findings of this study offer several practical implications.

No obvious benefits when using cross-model testing strategies. The idea of cross-model testing was motivated by the potential to prevent the same biases from affecting both implementation and tests. However, the results of this study did not show statistically significant improvements over same-model generation. Thus, for now, there is no reason to expect big benefits of cross-model testing strategies – at least in the context of Elixir.

Do not skip writing tests manually, yet. The manual test suites outperformed the generated test suites. While it took longer to create these tests (1-2 days) manually compared to generating a single test suite, it might still be worth doing. For 99 of the faulty implementations only the manual tests revealed implementation errors. The different LLMs only identified between 2 (GPT-4o-mini) and 20 (Llama-4-Maverick) erroneous implementations that were not flagged by test suites from other sources.

White-box test generation has drawbacks, too. While we decided for white-box testing in the pre-study to improve tests, we also observed some test errors that seem to be a result of that practice, e.g., generating tests for private functions. Generally, developers must be prepared to filter out unsuitable test suites or test cases before integrating them into products or systems.

LLMs need special support when working with concurrent code. Concurrency and process management seem to present particular challenges for LLMs, especially during test generation. Many generated tests failed because they incorrectly assumed synchronous responses from GenServer calls. In practice, developers should explicitly specify these patterns, expected timing behavior, and process lifecycle considerations in the prompts. Including example tests that correctly handle asynchronous calls may improve LLM output.

6.2 Comparison to Related Work

Similar to the works of Bhatia, Ryan, and Ouedraogo [4, 24, 21], our study confirms that, for Elixir too, LLMs are capable of generating test cases that are syntactically correct and achieve meaningful levels of code coverage, especially for simple and medium specifications. Here, test suites produced by models such as GPT-4.1 and Claude achieved high compilation success and coverage rates. We observe a compilation rate of around 88% for generated test suites, which is substantially higher than the rate (42.1%) reported by Yuan et al. for GPT-3.5-Turbo and Java. We attribute this to the more recent models used in our study and our prompt refinement upfront. Tarassow et al. found limitations of LLMs in working with domain-specific languages [30], specifically, issues with test logic occurred. This is similar to what we found for Elixir, which is also a language with fewer publicly available examples.

However, the study by Bhatia et al. reported many cases of incorrect assertions being generated. In their data roughly one third of assertions were semantically incorrect [4] and even the numbers of Nashid et al. [19] still included only 76% correct assertions. We found only 805 errors due to incorrect assertions in 2709 test suites with 28542 executions of single test cases, which points to a lower share of incorrect assertions. The difference may be explained by the different models used in the studies, as Bhatia and Nashid used GPT-3.5.

6.3 Threats to Validity

We discuss threats to internal, external, construct, and conclusion validity.

Internal Validity. We did not perform re-prompting to optimize prompts for the individual specifications. All calibration was done only in the pre-study. While this ensures comparability across all models, it may lead to an underestimation of the achievable performance of the LLMs [21, 24]. Test generation was performed using a white-box approach. While this approach likely allowed the LLMs to generate better tests, it also increased the possibility for false passes if the test copied a mistake made by the implementation, thus not finding it. Furthermore, ambiguities in the specifications might have directly influenced the quality and consistency of generated outputs. We performed the pre-study also with the goal of mitigating the risk of such ambiguities through a standardized structure. Flakiness in tests caused by mocking behavior and concurrency issues was observed in some specifications. These behaviors may have affected pass and fail counts and the interpretation of correctness for otherwise valid code or led to misinterpretation of faulty cases. Our analysis method did not allow us to exclude the possibility that implementation errors are present that were found by none of the tests. However, all of these internal issues affected all LLMs similarly. Thus, we still believe that our results allow a fair comparison of the scenarios using the same vs. different LLMs for implementation and test generation.

External Validity. Elixir and Phoenix tasks have likely a relatively low representation in current LLM training data, as little public code is available. As a result, the LLMs may struggle to generalize to the language-specific features or different patterns. While the nine specifications used cover a range of complexity levels and practical use cases, they do not represent a comprehensive sample of all possible Elixir programming tasks. For example, full Phoenix LiveView setups involving router configuration and multi-module contexts were excluded. This decision likely reduced compilation failures and the frequency of certain complex errors, but also limited the evaluation of LLMs' performance in high-complexity contexts. Thus, while our study provides first insights about LLMs' performance

in generating tests for Elixir programs, further studies are necessary. Moreover, as Elixir is a more niche programming language, there are fewer resources for LLM training which possibly lowers the performance of LLMs using this language compared to widely used programming languages like Java or Python. Therefore, the observed gap of human-written tests versus LLM-generated tests is likely larger for Elixir than would be the case for those languages. Also note that our results are merely a snapshot of LLMs' capabilities at the time of this study (April/May 2025). Newer models might be better suited for low-resource languages such as Elixir and therefore could exhibit higher rates of correct implementations and tests.

Construct Validity. We motivate our research by stating that using a different LLM for the test than for the implementation might be beneficial to rule out biases of models. However, by opting for white-box testing, some of the bias may have “spilled” to the model used for testing. However, as a white-box approach seems to be better when generating tests with LLMs [10], we accepted this validity concern in order to give a more realistic setting. Another concern is that the classification of errors into implementation or test faults required manual judgment, which always leaves room for errors. We performed the analysis in a systematic manner in order to reduce possible errors. Similarly, the human-written baseline implementations and tests were written by the first author without independent review. A more realistic baseline should comprise implementations and tests written by multiple humans to provide a more reliable comparison to LLM-generated artifacts.

Our conservative, deterministic generation setup (cf. Subsection 4.2) prioritizes comparability and reproducibility over matching typical usage; workflows involving best-of-n selection and iterative feedback would likely yield higher absolute performance, so our figures are best read as a lower bound. This limitation applies uniformly across models and conditions and thus does not affect our main comparison.

Conclusion Validity. The dataset is missing 5 test suites due to API errors or timeouts on the OpenRouter platform. The resulting five missing files represent only 0.16% of test executions and are unlikely to have had a measurable impact on the statistical outcomes. Other missing executions were due to implementation compilation failures and limited the number of test suite executions for test suites of all sources similarly.

7 Conclusion

We presented an experiment on code and test generation with LLMs for Elixir to identify whether generating code and tests with the same LLM decreases the chances of the test suite finding implementation faults. Our results do not indicate that this is the case. In addition, we found that our manually written baseline tests still identify more faulty implementations than all generated tests, despite being written as black-box tests, while the LLMs had access to the implementations. Unsurprisingly, the complexity of the task also negatively affects the correctness of generated implementations and tests. Our results also confirm that code coverage is not the best predictor for a test suite's ability to find errors. In future work, we plan to investigate whether this result can be replicated for other low-resource programming languages. Similarly, it would be interesting to also investigate cases where the LLMs need to generate multiple files or even consider complete repositories for solving the given tasks.

Data Availability Statement

All data is available on Zenodo [13]. The data includes implementations and test suites (manually written and generated) as well as the specifications and prompts used.

References

- 1 M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272, 2017. doi:10.1109/ICSE-SEIP.2017.27.
- 2 Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated Unit Test Improvement using Large Language Models at Meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024*, pages 185–196, New York, NY, USA, July 2024. Association for Computing Machinery. doi:10.1145/3663529.3663839.
- 3 Juan Altmayer Pizzorno and Emery D Berger. Coverup: Effective high coverage test generation for python. *Proceedings of the ACM on Software Engineering*, 2(FSE):2897–2919, 2025. doi:10.1145/3729398.
- 4 Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. 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, LLM4Code '24*, pages 54–61, New York, NY, USA, September 2024. Association for Computing Machinery. doi:10.1145/3643795.3648396.
- 5 Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs. *Artifact: Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs*, 8(OOPSLA2):295:677–295:708, October 2024. doi:10.1145/3689735.
- 6 Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024*, pages 572–576, New York, NY, USA, July 2024. Association for Computing Machinery. doi:10.1145/3663529.3663801.
- 7 Jacob Cohen. *Statistical power analysis for the behavioral sciences*. L. Erlbaum Associates, Hillsdale, N.J., 2nd ed edition, 1988.
- 8 Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. Effective test generation using pre-trained Large Language Models and mutation testing. *Information and Software Technology*, 171:107468, July 2024. doi:10.1016/j.infsof.2024.107468.
- 9 Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11*, pages 416–419, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2025113.2025179.
- 10 Alexandru Guzu, Georgian Nicolae, Horia Cucu, and Corneliu Burileanu. Large Language Models for C Test Case Generation: A Comparative Analysis. *Electronics*, 14(11):2284, January 2025. Number: 11 Publisher: Multidisciplinary Digital Publishing Institute. doi:10.3390/electronics14112284.
- 11 Sathvik Joel, Jie JW Wu, and Fatemeh H. Fard. A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages, November 2024. arXiv:2410.03981 [cs]. doi:10.48550/arXiv.2410.03981.
- 12 Philipp Kogler, Wei Chen, and Stefan Wallner. Code generation for niche programming languages with large language models. *Software Engineering 2025 – Companion Proceedings*, 2025. doi:10.18420/se2025-ws-13.
- 13 Noah Leu, Julian Oertel, and Regina Hebig. Replication package for "don't bother to use a second llm and write tests yourself!", May 2026. doi:10.5281/zenodo.20157738.

- 14 Yihao Li, Pan Liu, Haiyang Wang, Jie Chu, and W Eric Wong. Evaluating large language models for software testing. *Computer Standards & Interfaces*, 93:103942, 2025. doi:10.1016/J.CSI.2024.103942.
- 15 Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. No need to lift a finger anymore? assessing the quality of code generation by chatgpt. *IEEE Transactions on Software Engineering*, 50(6):1548–1584, 2024. doi:10.1109/TSE.2024.3392499.
- 16 Stephan Lukasczyk and Gordon Fraser. Pynguin: automated unit test generation for Python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ICSE '22, pages 168–172, New York, NY, USA, October 2022. Association for Computing Machinery. doi:10.1145/3510454.3516829.
- 17 Noble Saji Mathews and Meiyappan Nagappan. Test-Driven Development and LLM-based Code Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pages 1583–1594, New York, NY, USA, October 2024. Association for Computing Machinery. doi:10.1145/3691620.3695527.
- 18 Matthias M Müller. Two controlled experiments concerning the comparison of pair programming to peer review. *Journal of Systems and Software*, 78(2):166–179, 2005. doi:10.1016/J.JSS.2004.12.019.
- 19 Noor Nashid, Mifta Sintaha, and Ali Mesbah. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2450–2462, 2023. doi:10.1109/ICSE48619.2023.00205.
- 20 Julian Oertel, Jil Klünder, and Regina Hebig. Don't settle for the first! how many github copilot solutions should you check? *Information and Software Technology*, 183:107737, 2025. doi:10.1016/J.INFSOF.2025.107737.
- 21 Wendkuuni C. Ouedraogo, Kader Kabore, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawende F. Bissyande. LLMs and Prompting for Unit Test Generation: A Large-Scale Evaluation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pages 2464–2465, New York, NY, USA, October 2024. Association for Computing Machinery. doi:10.1145/3691620.3695330.
- 22 Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1297846.1297902.
- 23 Sujoy Roy Chowdhury, Giriprasad Sridhara, A K Raghavan, Joy Bose, Sourav Mazumdar, Hamender Singh, Srinivasan Bajji Sugumaran, and Ricardo Britto. Static Program Analysis Guided LLM Based Unit Test Generation. In *Proceedings of the 8th International Conference on Data Science and Management of Data (12th ACM IKDD CODS and 30th COMAD)*, CODS-COMAD '24, pages 279–283, New York, NY, USA, June 2025. Association for Computing Machinery. doi:10.1145/3703323.3703742.
- 24 Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM. *Proc. ACM Softw. Eng.*, 1(FSE):43:951–43:971, July 2024. doi:10.1145/3643769.
- 25 Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications, March 2025. arXiv:2402.07927 [cs]. doi:10.48550/arXiv.2402.07927.
- 26 Arkadii Sapozhnikov, Mitchell Olsthoorn, Annibale Panichella, Vladimir Kovalenko, and Pouria Derakhshanfar. Testspark: IntelliJ idea's ultimate test generation companion. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, pages 30–34, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3639478.3640024.

- 27 Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, January 2024. Conference Name: IEEE Transactions on Software Engineering. doi:10.1109/TSE.2023.3334955.
- 28 Rajendran Swamidurai, Brad Dennis, and Uma Kannan. Investigating the impact of peer code review and pair programming on test-driven development. In *IEEE SOUTHEASTCON 2014*, pages 1–5, March 2014. ISSN: 1558-058X. doi:10.1109/SECON.2014.6950664.
- 29 Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Transactions on Software Engineering*, 50(6):1340–1359, 2024. doi:10.1109/TSE.2024.3382365.
- 30 Artur Tarassow. The potential of LLMs for coding with low-resource and domain-specific programming languages, July 2023. arXiv:2307.13018 [cs]. doi:10.48550/arXiv.2307.13018.
- 31 Fadel Toure, Mourad Badri, and Luc Lamontagne. A metrics suite for JUnit test code: a multiple case study on open source software. *Journal of Software Engineering Research and Development*, 2(1):14, December 2014. doi:10.1186/s40411-014-0014-6.
- 32 András Vargha and Harold D. Delaney. A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000. Publisher: [American Educational Research Association, Sage Publications, Inc., American Statistical Association]. doi:10.2307/1165329.
- 33 Björn Walther. Kruskal-Wallis-Test in R rechnen, October 2022. URL: <https://bjoernwalther.com/kruskal-wallis-test-in-r-rechnen/>.
- 34 Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4):911–936, 2024. doi:10.1109/TSE.2024.3368208.
- 35 Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. On the Evaluation of Large Language Models in Unit Test Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE '24*, pages 1607–1619, New York, NY, USA, October 2024. Association for Computing Machinery. doi:10.1145/3691620.3695529.
- 36 Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and Improving ChatGPT for Unit Test Generation. *Proc. ACM Softw. Eng.*, 1(FSE):76:1703–76:1726, July 2024. doi:10.1145/3660783.
- 37 Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving chatgpt for unit test generation, 2024. doi:10.48550/arXiv.2305.04207.