Evaluating Large Language Models in Detecting Test Smells

Keila Lucas
Federal University of Campina Grande
Brazil
keila.santos@copin.ufcg.edu.br

Rohit Gheyi
Federal University of Campina Grande
Brazil
rohit@dsc.ufcg.edu.br

Elvys Soares
Federal Institute of Alagoas
Brazil
elvys.soares@ifal.edu.br

Márcio Ribeiro
Federal University of Alagoas
Brazil
marcio@ic.ufal.br

Ivan Machado
Federal University of Bahia
Brazil
ivan.machado@ufba.br

ABSTRACT

Test smells are coding issues that typically arise from inadequate practices, a lack of knowledge about effective testing, or deadline pressures to complete projects. The presence of test smells can negatively impact the maintainability and reliability of software. While there are tools that use advanced static analysis or machine learning techniques to detect test smells, these tools often require effort to be used. This study aims to evaluate the capability of Large Language Models (LLMs) in automatically detecting test smells. We evaluated ChatGPT-4, Mistral Large, and Gemini Advanced using 30 types of test smells across codebases in seven different programming languages collected from the literature. ChatGPT-4 identified 21 types of test smells. Gemini Advanced identified 17 types, while Mistral Large detected 15 types of test smells. The LLMs demonstrated potential as a valuable tool in identifying test smells.

KEYWORDS

Test Smells, Large Language Models (LLMs), Test Codes.

1 INTRODUCTION

The testing phase in software development is an important process to ensure the quality, functionality, and security of systems [31]. Researchers have coined the term test smells to describe potential design problems in test code, analogous to the code smells found in poorly designed source code [4, 33]. These symptoms can cause tests to exhibit erratic behavior, such as flakiness, false positives, and false negatives, compromising software quality due to their limited defect-catching capabilities. Despite the concept of test smells not being new, some studies have shown that they are prevalent in both open-source and industry projects, and they negatively impact code maintenance and understanding activities [3, 22, 34]. van Deursen et al. [33] and Meszaros [19] defined catalogs of test smells along with their refactoring actions.

There are various types of test smells, which can be characterized by code duplication (*Duplicated Assert*) [22], complexity in conditional structures, lack of documentation of assertions (*Assertion Roulette*), non-deterministic execution behavior [20], among others. Test smells can condition future difficulties in the maintenance process [28], as the incidence of test smells in the code base will impair the comprehensibility of the implemented structures as the software evolves [12, 13]. Listing 1 presents an example of the *Magic Numbers* test smell that occurs when a test method contains inexplicable and undocumented numeric values as parameters or values for identifiers. The numbers `15.5D`, `15` and `30` are magic numbers, since there is no indication of their semantics.

```
1 @Test
2 public void testGetLocalTimeAsCalendar() {
3   Calendar localTime = calc.getLocalTimeAsCalendar(
      BigDecimal.valueOf(15.5D), Calendar.getInstance());
4   assertEquals(15,localTime.get(Calendar.HOUR_OF_DAY));
5   assertEquals(30,localTime.get(Calendar.MINUTE));
6 }
```

Listing 1: Example code with *Magic Numbers Test.*

Recent studies [14, 21, 23, 27, 30] have investigated the impact of test smells [5] and how these indicators of poor coding affect the comprehension and quality of software [31]. Existing tools that detect test smells often rely on advanced static analysis or machine learning techniques, which can be complex to implement and extend with new smells, as well as support multiple languages [2]. Moreover, these tools require effort to use. Only a few provide explanations and suggestions for code improvements.

Artificial Intelligence (AI) techniques, particularly Large Language Models (LLMs), offer the potential to improve the test review process [35]. LLMs have revolutionized natural language processing, demonstrating remarkable performance across various tasks, including question answering, machine translation, and text generation [35]. These models have also impacted various domains within software engineering [10], expanding the possibilities of integrating natural language analysis capabilities to enhance source code analysis, including detecting test smells. However, to the best of our knowledge, no study so far indicates to what extent LLMs can help detect test smells.

In this paper, we evaluate the capability of LLMs in the automatic detection of test smells. We evaluated ChatGPT-4, Mistral Large, and Gemini Advanced. These models were tested on 30 types of test smells across codebases in seven different programming languages, which were collected from the existing literature [1]. ChatGPT-4 detected 21 types of test smells. Gemini Advanced identified 17 types, while Mistral Large detected 15 types of test smells. The results indicate that LLMs demonstrate efficiency in detecting test smells. ChatGPT-4, in particular, can identify 70% of the test smell types in the code, suggesting that LLMs could be a valuable tool in enhancing the quality of software by identifying and mitigating test smells. The LLMs demonstrated potential as valuable tools to be integrated into IDEs for the detection of test smells. Their processing capabilities enable the detection and explanation of various types of test smells and offer suggestions for code improvements.

2 METHODOLOGY

2.1 Research Questions

Our goal is to evaluate ChatGPT-4, Gemini Advanced 1.5, and Mistral Large concerning detecting test smells from the point of view of researchers in the context of test smells cataloged in the literature. We address the following research questions:

- **RQ**$_1$ To what extent can ChatGPT-4 detect test smells?
- **RQ**$_2$ To what extent can Gemini Advanced detect test smells?
- **RQ**$_3$ To what extent can Mistral Large detect test smells?

We compare the output of each LLM with the test smells presented in the catalog [1].

2.2 Planning

The Open Catalog of test smells [1] provides a dataset of 127 formal and informal sources, featuring various types of test smells, identified by name, Also Known as (AKA) when available, conceptual definition, and code examples, as well as citations of references for further literature review. The catalog consists of six categories of test smells: Code Related, Dependencies, Design Related, Issues in Test Steps, Test Execution – behavior, and Test Semantic – logic.

We evaluated a set of 30 test smells from the catalog [1]. Each test smell is illustrated with at least one small example, some of which are extracted from real projects [9, 11, 30]. In eleven cases, there are ⟨`code snippets`⟩ of unit tests implemented using the JUnit framework. Three cases correspond to complete programs along with the test class. In four cases, the examples consist solely of the test class. Meanwhile, in the remaining twelve cases, the examples are snippets of individual methods or functions. Some types (i.e., *Assertion Roulette*, *Duplicate Assert*, *Exception Handling*, and *Test Code Duplication*) were considered due to their frequent occurrence in popular open-source projects [30].

For the queries, we use zero-shot prompting [7, 16, 24], which refers to a query scenario in which the machine learning model receives a task for which it has not been explicitly trained to perform. The model uses the general understanding of the submitted query and the pre-existing knowledge to perform the task without any additional adjustments or specific examples related to the task in question. We use the following prompt in each LLM with default parameters:

- Consider the following ⟨`language`⟩ test case. Does it have any test smells? ⟨`code snippet`⟩

We evaluate test smells in the following ⟨`language`⟩: C#, Java, JavaScript, Python, Ruby, Smalltalk and TTCN-3. The consultation with the LLMs occurred in May 2024.

3 RESULTS

The LLMs showed promising results in identifying test smells in source code. ChatGPT-4 demonstrated the best performance in detecting test smells, successfully identifying 21 out of 30 types, with the misses being the *Bad Comment Rate*, *Duplicated Code In Conditional*, *Duplicate Statements*, *Irrelevant Information* and *Overcommented Test* test smells. It partially detects the *Badly Used Fixture*, *Constant Actual Parameter Value*, *Exception Handling*, and *Two For The Price Of One* test smells. The Gemini detect 17 test smells and three partially detects, while Mistral detect 15 test smells and five partially detects. At least one LLM can detect 25 out of 30 test smells. The *Overcommented Test* smell in Smalltalk and the *Duplicated Code In Conditional* in TTCN-3 are not detected by the evaluated LLMs in our study.

Table 1 presents the detailed performance of LLMs in identifying the different types of test smell, with highlights for correct identifications (✓), partial correct identifications (◐), errors (×), language, and lines of code (LOC). Partial hits are characterized when the LLM returns information related to the definition of the test smell but identifies it as a different type of test smell, which has a similar definition, or when the presented information is too simple, without details for the example presented.

4 DISCUSSION

4.1 Number of Attempts

Due to the probabilistic nature of their processing, the models can produce varied responses to the same query, even when the same prompt is applied [25]. For the test smells not identified in the first submission, we execute two additional attempts (2^{nd} and 3^{rd}) for each specific example, using a unique chat session for each round with the same prompt. When the LLM succeeded on the second attempt, a third attempt was not necessary. The types of test smells that were detected on the first query attempt were not subjected to additional attempts. All correct responses obtained in the outputs were reviewed by two authors to formally record the performance of each LLM. In the additional attempts (2^{nd} and 3^{rd}), no feedback was provided on the results presented, all additional attempts were made by resetting the conversation window, and applying the same prompt. Table 2 presents the overall summary of the attempts executed on ChatGPT-4, Gemini Advanced, and Mistral Large.

In response to our **RQ**$_1$, the results of the analysis confirm that ChatGPT-4 demonstrated the best detection performance over 3 attempts, identifying 26 out of 30 examples in total. ChatGPT-4 was unable to detect the *Duplicated Code in Conditional* and *Overcommented Test* test smells, and only partially detected the *Constant Actual Parameter Value* and *Two for the Price of One* test smells. The results regarding the performance of Gemini Advanced address our **RQ**$_2$, in which we highlight that the LLM identified 17 types of test smells on the first query attempt. In additional attempts (2^{nd} and 3^{rd}), Gemini Advanced showed improvement by identifying seven more types of the 13 that were not correctly identified in the first attempt. Table 3 presents the attempts for the test smells re-evaluated in Gemini Advanced.

Mistral Large had the weakest performance in the test smell detection process, and in response to our **RQ**$_3$, we observed that the LLM correctly detected only 15 types of test smells on the first query attempt. In the additional attempts (2^{nd} and 3^{rd}), the LLM was able to identify six more types of test smells that were previously undetected, specifically: *Expected Exceptions And Verification*, *Hidden Test Call*, *Obscure Test*, *Plate Spinning*, *Self-Test*, and *The First And Last Rites*.

All LLMs do not detect the *Duplicated Code In Conditional* test smell (see Listing 2) in 3 attempts. ChatGPT detected other types of issues in the functions `checkSomething` and `checkSomethingElse`,

Table 1: LLMs performance summary.

ID	Test Smell	Language	LOC	ChatGPT-4	Gemini Advanced	Mistral Large
1	Anonymous Test	Java	4	✓	✓	✓
2	Assertion Roulette	Python	20	✓	◐	✓
3	Asynchronous Test	Java	17	✓	✓	✓
4	Bad Comment Rate	TTCN-3	52	×	×	✓
5	Badly Used Fixture	Java	24	◐	◐	✓
6	Constant Actual Parameter Value	TTCN-3	13	◐	×	◐
7	Context Logic In Production Code	Java	7	✓	✓	✓
8	Duplicate Assert	Java	24	✓	✓	✓
9	Duplicated Code In Conditional	TTCN-3	22	×	×	×
10	Duplicate Statements	TTCN-3	9	×	✓	✓
11	Empty Test	Java	5	✓	✓	✓
12	Exception Handling	Java	30	◐	✓	×
13	Expected Exceptions And Verification	Java	8	✓	✓	×
14	Fire And Forget	Ruby	22	✓	✓	◐
15	Hard-Coded Test Data	Java	9	✓	×	✓
16	Hidden Test Call	C#	17	✓	✓	◐
11	Irrelevant Information	Java	11	×	×	◐
18	Long Test	Ruby	41	✓	✓	✓
19	Magic Number Test	Java	6	✓	×	✓
20	Obscure Test	Ruby	18	✓	×	×
21	Overcommented Test	Smalltalk	21	×	×	×
22	Overspecified Tests	Java	30	✓	✓	✓
23	Plate Spinning	JavaScript	50	✓	✓	◐
24	Redundant Print	Java	9	✓	✓	✓
25	Self Important Test Data	Ruby	67	✓	✓	×
26	Self-Test	Java	13	✓	×	×
27	Sensitive Equality	Java	12	✓	◐	×
28	Test Code Duplication	Python	18	✓	✓	✓
29	The First And Last Rites	Java	13	✓	✓	✓
30	Two For The Price Of One	Java	13	◐	×	×
			Total	21	17	15

✓hits ◐ partial hits × errors

Table 2: Number of test smells identified per attempt for each LLM.

LLM	1^{st}	2^{nd}	3^{rd}	Total
ChatGPT-4	21/30	3/9	2/6	26/30
Gemini Advanced	17/30	5/13	2/8	24/30
Mistral Large	15/30	5/15	1/10	21/30

for example, *Magic Numbers*, *Multiple Exit Points*, *External Dependencies* and *Implicit Else Condition*. In each attempt, the ChatGPT output shows little variation in responses.

```
function checkSomething(in float p1, in float p2) return
    boolean {
    if (p1 < 0.0) {
        return false;
    } if (p2 >= 7.0) {
        return false;
    } ...
}
function checkSomethingElse(in float p1) runs on
    ExampleComponent {
    var charstring result;
    if (p1 > 0) {
        result := "foo";
        pt.send(result);
    } else {
        result := "bar";
        pt.send(result);
    }
}
```

Listing 2: Example code with *Duplicated Code In Conditional*.

Table 3: Gemini attempts.

Test Smell	1^{st}	2^{nd}	3^{rd}
Assertion Roulette	◐	×	✓
Bad Comment Rate	×	◐	✓
Badly Used Fixture	◐	✓	
Constant Actual Parameter Value	×	×	×
Duplicated Code In Conditional	×	×	×
Hard-Coded Test Data	×	✓	
Irrelevant Information	×	×	×
Magic Number Test	×	✓	
Obscure Test	×	✓	
Overcommented Test	×	×	×
Self-Test	×	×	×
Sensitive Equality	◐	✓	
Two For The Price Of One	×	×	×

4.2 Detecting Test Smells

Six test smells are detected by only one LLM in the first attempt: *Bad Comment Rate* (Mistral), *Badly Used Fixture* (Mistral), *Exception Handling* (Gemini), *Obscure Test* (ChatGPT), Self-Test (ChatGPT) and *Sensitive Equality* (ChatGPT), as presented in Table 1. Four test smells are not detected by any LLM (the *Constant Actual Parameter Value*, *Duplicated Code In Conditional*, *Overcommented Test*, and *Two For The Price Of One* test smells).

LLMs also suggest other test smells in the code snippets evaluated. Some responses from the LLMs did not exactly match the label of the test smell recorded in the catalog [1] but provided related and coherent explanations. For instance, the *Anonymous Test* smell occurs when a test whose name is meaningless as it does not express the purpose of the test in the current context [1]. Consider the example presented in Listing 3. ChatGPT-4 yields the following output considering it as the `Inadequate Test Method Name` test smell:

- "The name of the test method, `test1`, is not descriptive. It does not convey what aspect of the `LoginPage.login` method it is testing. Good test names should describe the purpose of the test and the specific behavior or scenario being tested."

Since the test smells definitions are similar, we consider that the ChatGPT-4 answer is correct. In some cases, LLMs also suggest refactorings to remove test smells, such as providing better names. ChatGPT-4 also indicates other problems in the test case. For example, it indicates that the test case does not have assertions.

```
@Test
public void test1() {
  LoginPage.login("user", "password");
}
```

Listing 3: Example code with *Anonymous Test*.

The *Assertion Roulette* test smell occurs when a test method has multiple non-documented assertions. It is not possible to understand the reason for the failure of the test [1]. Listing 4 presents part of the code snippet containing this test smell. ChatGPT-4 and Mistral detected this test smell on the first attempt, while Gemini provided a partially correct response, indicating that there is an "Assertion Overload" in the code example. Mistral correctly yields the following explanation:

- "Lack of Assertion Messages: The test case does not provide any custom assertion messages. This can make it harder to understand why a test failed when it does. Providing meaningful assertion messages can make the test results more readable."

In the first two attempts, Gemini does not complain about excessive assertions. However, in the third attempt, it identifies the test smell:

- "Excessive Assertion: The test asserts multiple things at once (`mileage`, `fullFuel`, `isValidAirLineCode`). This violates the one assertion per test principle, making it harder to pinpoint the cause of a failure."

```
class TestFlight(unittest.TestCase):
  def test_flight(self):
    flight = Flight('2569',1000)
    self.assertEqual(flight.mileage,1000)
    self.assertTrue(flight.fullFuel)
    self.assertTrue(flight.isValidAirLineCode())
```

Listing 4: Example code with *Assertion Roulette*.

The *Exception Handling* smell occurs when a test method explicitly passes or fails and is dependent on the production method throwing an exception. Developers should utilize JUnit's exception handling to automatically pass/fail the test instead of writing custom exception handling code or throwing an exception. Listing 5 presents part of the code snippet containing this test smell. Gemini correctly detects it in the first attempt:

- "Lack of Error Handling Assertions: The test does not verify the correct handling of error conditions or exceptional cases, which can leave gaps in test coverage."

ChatGPT-4 partially detects the problem in the `Assert.fail` but it does not provide a good explanation:

- "Assertion Messages: The test uses `Assert.fail(e.getMessage());` within a try-catch block to handle exceptions, which is a valid approach, but the test assertions themselves do not include custom failure messages. This could make it harder to diagnose issues because the output will not explain what the expected logic or outcome was supposed to be, only that the expected and actual values did not match."

However, it correctly identifies the test smell in the third attempt:

- "Error Handling in Tests: The use of try-catch block with `Assert.fail()` inside the catch is a valid technique for ensuring exceptions are not expected. However, this could be improved by using `@Test(expected = CalculationException.class)` if the intention was to assert that an exception should not be thrown. If the test's intention is to handle an unexpected exception, then it is generally set up correctly."

Mistral does not explain any issues related to `Assert.fail` in three attempts.

```
@Test
public void realCase() {
  Point p34 = new Point("34", ...); ...
  try {
    a.compute();
  } catch (CalculationException e) {
    Assert.fail(e.getMessage());
  } ...
}
```

Listing 5: Example code with *Exception Handling*.

4.3 Programming Languages

The accuracy rate of the LLMs for detecting different types of test smells across various programming languages was evaluated based on the results presented in Table 4. Seventeen code examples submitted to the LLMs are written in Java. ChatGPT-4 stands out by correctly identifying 13 (76%) of the 17 types of test smells analyzed. For other languages, such as C#, JavaScript, Python, Ruby, and TTCN-3, the LLMs demonstrated varying degrees of success. For C#, ChatGPT-4 and Gemini Advanced detected the *Hidden Test Call* test smell, respectively. In Python, Mistral Large and ChatGPT-4 achieved a 100% detection rate (2/2), while Gemini Advanced

correctly identified 1 out of 2 test smells. For JavaScript, Mistral Large is unable to identify the type of smell in the example, but ChatGPT and Gemini were successful. For Ruby, ChatGPT-4, Gemini Advanced, and Mistral Large identified 4/4, 3/4, and 1/4 test smells, respectively. For the TTCN-3 language, ChatGPT-4 failed to identify the *Bad Comment Rate*, *Duplicated Code In Conditional*, *Duplicate Statements* test smells on the first attempt. The model achieved only a partial success for the *Constant Actual Parameter Value* test smell, indicating "Similar test data" for the function `f` that sends messages with almost identical data in two cases, but it was not specific about the use of parameters in the test.

All three models failed to detect the *Overcommented Test* smell in three attempts. No LLM was able to identify the issue of excessive comments in Smalltalk, which obscure the code and divert attention from the purpose of the test. The responses either did not reference problems related to comments or diverged from the definition of the submitted test smell type, indicating a detection error. But they detected other issues, such as the *Mystery Guest*, *Conditional Complexity*, *Test Dependencies*, and *Eager Test* smells.

Table 4: Number of test smells identified by programming language for each LLM.

Language	GPT-4	Gemini Advanced	Mistral Large
C#	1/1	1/1	0/1
Java	13/17	10/17	10/17
JavaScript	1/1	1/1	0/1
Python	2/2	1/2	2/2
Ruby	4/4	3/4	1/4
Smalltalk	0/1	0/1	0/1
TTCN-3	0/4	1/4	2/4

4.4 Metamorphic Testing

Metamorphic testing (MT) is an active area of research focused on improving model robustness [6]. This testing approach involves generating new data samples (code) by applying metamorphic transformations to the original validation or testing data. The modifications applied to the code snippets are controlled to maintain semantic and behavioral equivalence with the original code, while developing modifications in their Abstract Syntax Trees (ASTs), considering the restructuring of variables, parameters, and methods, as well as the removal of comments. The purpose of this methodology is to test the resilience of models, allowing the identification of hidden faults that may not be apparent in the original evaluation.

We select 10 test smells of different sizes and languages. The selected examples are syntactically modified to be submitted in new queries to the LLMs. For example, we changed variable names, method names, some numeric parameters, and declared strings. The modification made to the original code (Listing 6) of the *Context Logic In Production Code* test smell is presented in Listing 7.

```
public static void SaveToDatabase(Customer
    customerToWrite) {
  if(AreWeTesting)
    WriteWithMockDatabase(customerToWrite);
  else
    Write(customerToWrite);
}
```

Listing 6: Original code.

```
public static void TestBD(Customer customerTW) {
  if(makingTest)
    WriteWithMockDatabase(customerTW);
  else
    Write(customerTW);
}
```

Listing 7: Listing 6 modified for the metamorphic test.

Table 5 presents the performance of the models. ChatGPT-4 again achieved the best performance, correctly identifying all examples, even for the *Duplicate Statements* test smell that was detected in the second attempt in the original code. The MT allowed Mistral to correctly detect the test smell *Plate Spinning*, which had previously been detected only partially, as the LLM did not indicate that the test could fail before the calls were completed. After the submission of the MT code, the model identifies that the test can fail due to the unpredictability of external requests.

Gemini correctly detects 4 out of 10 test smells in the MT. It fails to detect the *Duplicate Statements* test smell, not being able to identify the repeated structures. The LLM highlights other issues such as "Obscure Intent", "Conditional Test Logic", and "Potential Assertion Roulette", but it does not emphasize problems related to the repetition of structures in the original code. Gemini occasionally misidentified certain types of test smells. The LLM presented the *General Fixture* test smell as the answer for the *Long Test* and *The First And Last Rites* test smells that were identified in the original code. The confusion between the types *The First And Last Rites* and *General Fixture* may have been caused by the similarity between the definitions of both types. As both types involve problems with the occurrence of recurring structures, the LLM may have interpreted them as synonymous types of smells.

4.5 Threats to Validity

There are some threats to validity that could impact the results and interpretations [25]. The test smells may be part of the LLMs training data. We conducted metamorphic testing to reduce this threat to validity. The way prompts are structured may influence the responses, potentially making them more general for types that require specific explanations. This could affect the accuracy and specificity of the test smell detection. We use a simple prompt. It is not always an easy task to check whether the LLM output is aligned with the test smell definition. We checked each answer with two authors of the paper.

Most test smells are evaluated using a single test case example. The limited number of examples per test smell could affect the robustness of the findings. Additionally, some examples are separated from the original class context, which might not fully represent real-world scenarios. The study focused on a specific set of test smells and evaluated examples primarily in Java, with fewer examples from other languages. This limited diversity may affect the generalizability of the findings to other programming languages and test smell types. Moreover, the study's findings might not apply to other codebases or test cases not represented in the catalog. The reproducibility of the study results is dependent on the consistent behavior of LLMs, which may vary with updates or changes in the models.

Table 5: Metamorphic Testing summary.

	ChatGPT-4		Gemini Advanced		Mistral Large	
Test Smell	Original	MT	Original	MT	Original	MT
Anonymous Test	✓	✓	✓	◐	✓	✓
Asynchronous Test	✓	✓	✓	✓	✓	✓
Context Logic In Production Code	✓	✓	✓	✓	✓	✓
Duplicate Statements	×	✓	✓	×	✓	✓
Fire And Forget	✓	✓	✓	◐	◐	◐
Hard-Coded Test Data	✓	✓	×	×	✓	✓
Long Test	✓	✓	✓	◐	✓	✓
Plate Spinning	✓	✓	✓	✓	◐	✓
Test Code Duplication	✓	✓	✓	✓	✓	✓
The First And Last Rites	✓	✓	✓	◐	×	◐
Total	9	10	9	4	7	8

5 RELATED WORK

Pontillo et al. [23] proposed a method based on machine learning (ML) to detect test smells, focusing on four specific test smells: *Eager Test, Mystery Guest, Resource Optimism*, and *Test Redundancy*, aiming to overcome the limitations of existing heuristic techniques. The authors applied the ML approach to predict the likelihood of a test case being affected by a specific smell. They will also compare their method with established heuristic techniques.

Soares et al. [29] aimed to investigate developers' awareness of test smells' existence and acceptance of their refactorings in submitted pull requests. The authors demonstrate that developers are not always acquainted with the terminology of test smells but recognize their effects and harmfulness when consulted. Soares et al. [30] conducted a mixed-methods analysis involving 485 Java projects, investigating the extent to which developers adopt JUnit 5 and its new features to enhance test code quality. The study identified JUnit 5 features that can help remove test smells, such as *Assertion Roulette*, *Test Code Duplication*, and *Conditional Test Logic*.

Aljedaani et al. [2] compiled a catalog of test smell detection tools. Lambiase et al. [15] proposed an IntelliJ plugin called DARTS (Detection and Refactoring of Test Smells) that detects the *Eager Test*, *General Fixture*, and *Lack of Cohesion of Test Methods* test smells. Another tool is the RTj framework by Martinez et al. [18] that deals with *Rotten Green Test Cases*, which are tests that pass despite having at least one unexecuted assertion. Although RTj's test smells are outside our scope, it offers refactoring actions limited to substituting a failing assertion with a call to the `fail` method and adding a `TODO` comment to the problematic code. Santana et al. [26] proposed RAIDE, which is an open-source, IDE-integrated tool that addresses the *Assertion Roulette* and *Duplicated Assert* test smells in Java projects.

De Bleser et al. [8] assessed the diffusion and perception of test smells in SCALA projects, finding low diffusion of test smells across SCALA test classes and that many developers struggle to correctly identify most smells, despite recognizing design issues. Peruma et al. [22] investigated test smells in open-source Android applications and found that developers generally recognize the proposed smells as bad programming practices in unit test files.

Junior et al. [13] investigated the causes of test smell introduction by developers, revealing that even experienced professionals introduce test smells during their daily programming tasks despite using standardized company practices. Another study by Spadini et al. [32] examined the severity rating of four test smells and their perceived impact on test suite maintainability. They found that developers consider current detection rules for specific test smells too strict and that the newly defined severity thresholds align with participants' perceptions of how test smells impact test suite maintainability. Yang et al. [36] identified and defined new test smell types from software practitioners' discussions on Stack Overflow and developed a detector to identify these smells in real-world Java projects. Through empirical evaluation and practical validation, the study demonstrated the prevalence and impact of these test smells, providing insights for improving test code quality.

In our work, we investigate the extent to which LLMs can be useful for detecting test smells in the source code of software test cases. ChatGPT-4 successfully detects 70% of the test smells, showing promising results. In future work, we intend to evaluate to what extent LLMs can help in refactoring test smells.

6 CONCLUSION

In this paper, we evaluated the potential of LLMs for detecting test smells in source code. The LLMs demonstrated competence in detecting various types of test smells, proving to be an important auxiliary resource for software testing tasks. Notably, ChatGPT-4 achieved the highest overall accuracy rate of 70%. All data from this work are available online [17]. The results indicate opportunities to enhance tools for detecting test smells. By integrating LLMs into the software development lifecycle, developers can more effectively identify and address test smells. The automation of test smell detection using LLMs reduces the manual effort required in the testing phase. Further research and development are needed to improve the robustness and accuracy of LLMs in detecting a wider range of test smells, particularly in less common programming languages and more nuanced test smell types.

In future work, we intend to deepen the analysis of test smell types by considering a larger number of examples in test cases across various programming languages. This approach will allow for a more robust evaluation of the performance of ChatGPT-4, Mistral, and Gemini. Additionally, we plan to expand the review to include code bases from open-source projects and to investigate the potential of other LLMs, such as ChatGPT4-o, Claude, and Llama. Gemini has a 1 million-token context window, which enables it to evaluate larger programs.

REFERENCES

[1] 2024. The Open Catalog of Test Smells. https://test-smell-catalog.readthedocs.io/en/latest/.

[2] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D. Newman, Abdullatif Ghallab, and Stephanie Ludi. 2021. Test Smell Detection Tools: A Systematic Mapping Study. In *International Conference on Evaluation and Assessment in Software Engineering*. 170–180. https://doi.org/10.1145/3463274.3463335

[3] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2015. Are test smells really harmful? An empirical study. *Empirical Software Engineering* 20, 4 (2015), 1052–1094. https://doi.org/10.1007/s10664-014-9313-0

[4] K. Beck. 2003. *Test-driven development: by example*. Addison-Wesley Professional.

[5] Denivan Campos, Larissa Rocha, and Ivan Machado. 2021. Developers perception on the severity of test smells: an empirical study. *arXiv preprint arXiv:2107.13902* (2021).

[6] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *Computing Surveys* 51, 1 (2018), 4:1–4:27.

[7] DAIR.AI. 2024. Prompt Engineering Guide. https://www.promptingguide.ai/techniques.

[8] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. 2019. Assessing Diffusion and Perception of Test Smells in Scala Projects. In *International Conference on Mining Software Repositories*. 457–467. https://doi.org/10.1109/MSR.2019.00072

[9] Benedikt Hauptmann, Maximilian Junker, Sebastian Eder, Lars Heinemann, Rudolf Vaas, and Peter Braun. 2013. Hunting for smells in natural language tests. In *International Conference on Software Engineering*. IEEE Computer Society, 1217–1220.

[10] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. arXiv:2308.10620 [cs.SE]

[11] Manoel Aranda III, Naelson Oliveira, Elvys Soares, Márcio Ribeiro, Davi Romão, Ullyanne Patriota, Rohit Gheyi, Emerson Souza, and Ivan Machado. 2024. A Catalog of Transformations to Remove Smells From Natural Language Tests. In *International Conference on Evaluation and Assessment in Software Engineering*. ACM, 7–16.

[12] Nildo Silva Junior, Luana Martins, Larissa Rocha, Heitor Costa, and Ivan Machado. 2021. How are test smells treated in the wild? A tale of two empirical studies. *Journal of Software Engineering Research and Development* 9 (2021), 9–1.

[13] Nildo Silva Junior, Larissa Rocha, Luana Almeida Martins, and Ivan Machado. 2020. A survey on test practitioners' awareness of test smells. In *Iberoamerican Conference on Software Engineering*. Curran Associates, 462–475.

[14] Dong Jae Kim, Tse-Hsun Chen, and Jinqiu Yang. 2021. The secret life of test smells-an empirical study on test smell evolution and maintenance. *Empirical Software Engineering* 26 (2021), 1–47.

[15] Stefano Lambiase, Andrea Cupito, Fabiano Pecorelli, Andrea De Lucia, and Fabio Palomba. 2020. Just-In-Time Test Smell Detection and Refactoring: The DARTS Project. In *International Conference on Program Comprehension*. 441–445. https://doi.org/10.1145/3387904.3389296

[16] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Computing Surveys* 55, 9 (2023), 1–35.

[17] Keila Lucas, Rohit Gheyi, Elvys Soares, Márcio Ribeiro, and Ivan Machado. 2024. Evaluating Large Language Models in Detecting Test Smells (artifacts). https://zenodo.org/records/12748511.

[18] Matias Martinez, Anne Etien, Stéphane Ducasse, and Christopher Fuhrman. 2020. RTj: A Java Framework for Detecting and Refactoring Rotten Green Test Cases. In *International Conference on Software Engineering: Companion Proceedings*. 69–72. https://doi.org/10.1145/3377812.3382151

[19] G. Meszaros. 2007. *xUnit test patterns: Refactoring test code*. Pearson Education.

[20] Fabio Palomba and Andy Zaidman. 2020. Retraction Note: Retraction note to: The smell of fear: on the relation between test smells and flaky tests. *Empirical Software Engineering* 25, 4 (2020), 3041.

[21] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J Hellendoorn. 2022. Test smells 20 years later: detectability, validity, and reliability. *Empirical Software Engineering* 27, 7 (2022), 170.

[22] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2019. On the Distribution of Test Smells in Open Source Android Applications: An Exploratory Study. In *International Conference on Computer Science and Software Engineering*. 193–202.

[23] Valeria Pontillo, Dario Amoroso d'Aragona, Fabiano Pecorelli, Dario Di Nucci, Filomena Ferrucci, and Fabio Palomba. 2024. Machine learning-based test smell detection. *Empirical Software Engineering* 29, 2 (2024), 1–44.

[24] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners.

[25] June Sallou, Thomas Durieux, and Annibale Panichella. 2024. Breaking the Silence: the Threats of Using LLMs in Software Engineering. In *International Conference on Software Engineering - New Ideas and Emerging Results*. ACM/IEEE.

[26] Railana Santana, Luana Martins, Larissa Rocha, Tássio Virgínio, Adriana Cruz, Heitor Costa, and Ivan Machado. 2020. RAIDE: A Tool for Assertion Roulette and Duplicate Assert Identification and Refactoring. In *34th Brazilian Symposium on Software Engineering (SBES)*. 374–379. https://doi.org/10.1145/3422392.3422510

[27] Railana Santana, Luana Martins, Tássio Virgínio, Larissa Rocha, Heitor Costa, and Ivan Machado. 2024. An empirical evaluation of RAIDE: A semi-automated approach for test smells detection and refactoring. *Science of Computer Programming* 231 (Jan. 2024), 103013. https://doi.org/10.1016/j.scico.2023.103013

[28] Elvys Soares, Manoel Aranda III, Naelson Oliveira, Márcio Ribeiro, Rohit Gheyi, Emerson Souza, Ivan Machado, André L. M. Santos, Baldoino Fonseca, and Rodrigo Bonifácio. 2023. Manual Tests Do Smell! Cataloging and Identifying Natural Language Test Smells. In *International Symposium on Empirical Software Engineering and Measurement*. IEEE, 1–11.

[29] Elvys Soares, Márcio Ribeiro, Guilherme Amaral, Rohit Gheyi, Leo Fernandes, Alessandro Garcia, Baldoino Fonseca, and André Santos. 2020. Refactoring Test Smells: A Perspective from Open-Source Developers. In *Brazilian Symposium on Systematic and Automated Software Testing*. 50–59. https://doi.org/10.1145/3425174.3425212

[30] Elvys Soares, Márcio Ribeiro, Rohit Gheyi, Guilherme Amaral, and André L. M. Santos. 2023. Refactoring Test Smells With JUnit 5: Why Should Developers Keep Up-to-Date? *IEEE Transactions on Software Engineering* 49, 3 (2023), 1152–1170.

[31] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On the relation of test smells to software code quality. In *International conference on software maintenance and evolution*. IEEE, 1–12.

[32] Davide Spadini, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, and Alberto Bacchelli. 2020. Investigating Severity Thresholds for Test Smells. In *International Conference on Mining Software Repositories (MSR)*. 311–321. https://doi.org/10.1145/3379597.3387453

[33] Arie van Deursen, Leon Moonen, Alex van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In *International conference on extreme programming and flexible processes in software engineering*. 92–95.

[34] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. 2007. On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test. *IEEE Transactions on Software Engineering* 33, 12 (2007), 800–817. https://doi.org/10.1109/TSE.2007.70745

[35] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* 50 (2024), 911–936.

[36] Yanming Yang, Xing Hu, Xin Xia, and Xiaohu Yang. 2024. The Lost World: Characterizing and Detecting Undiscovered Test Smells. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024). https://doi.org/10.1145/3631973