Can LLMs Detect and Refactor Test Smells in Performance Tests?

Adelson Ornellas
Computing Institute, Federal University of Alagoas
Maceió, Brazil
ajof@ic.ufal.br

Manoel Aranda
Computing Institute, Federal University of Alagoas
Maceió, Brazil
mpat@ic.ufal.br

Márcio Ribeiro
Computing Institute, Federal University of Alagoas
Maceió, Brazil
marcio@ic.ufal.br

Samira Silva
Institute of Science, Technology and Innovation, Federal University of Lavras
São Sebastião do Paraíso, Brazil
samirasilva@ufla.br

Rohit Gheyi
Center for Electrical and Computer Engineering, Federal University of Campina Grande
Campina Grande, Brazil
rohit@dsc.ufcg.edu.br

Baldoino Fonseca
Computing Institute, Federal University of Alagoas
Maceió, Brazil
baldoino@ic.ufal.br

Ivan Machado
Computing Institute, Federal University of Bahia
Salvador, Brazil
ivanmachado@ufba.br

Abstract

Performance testing is essential to ensure that software meets its responsiveness and scalability requirements, and tools such as k6 are used to write performance test scripts that exercise these aspects in a structured and optimized way. Like any code, regardless of the programming language, k6 scripts are subject to recurring bad practices, known as performance test smells, that can yield misleading results or waste resources. While Large Language Models (LLMs) have shown promise in detecting and refactoring conventional test smells, their effectiveness on performance test smells remains largely unexplored. This paper evaluates whether three widely used LLMs (Claude, GPT, and Gemini) can detect and refactor four performance test smells in real-world k6 scripts, and whether adding illustrative code examples to the prompt improves their performance. Using a dataset of 228 k6 scripts collected from 58 GitHub repositories and labeled by a static linter, we submitted 1,368 detection-and-refactoring queries across two prompt variants and manually reviewed 120 of the resulting refactorings. We find that detection performance is moderate and comparable across the three models (F1 micro between 0.49 and 0.54, and between 0.53 and 0.54 under the definition-only prompt), that adding code examples does not consistently improve detection and can even degrade it, and most notably that detection difficulty and refactoring difficulty are largely decoupled: the smells that are hardest to detect are often refactored almost flawlessly once found, whereas one of the most reliably detected smells accounts for nearly all incorrect and breaking refactorings. We attribute this asymmetry to a distinction between declarative and architectural fixes, and discuss its implications for trusting LLM-generated refactorings in performance testing.

Keywords

Performance Test Smells, Large Language Models, k6

1 Introduction

Performance testing is a critical activity in modern software development: it verifies whether a system meets its responsiveness, scalability, and resource-usage requirements under load, and helps reveal bottlenecks before they reach production [7]. To support this activity, load testing frameworks such as k6 let developers express performance tests as scripts that are versioned, reviewed, and integrated into CI/CD pipelines [6]. As a result, these scripts have become first-class software artifacts and, like any code, are exposed to quality problems.

A well-established way to reason about such quality problems is the notion of test smells: symptoms in test code that may indicate deeper design or maintainability issues [18]. The concept was introduced by van Deursen et al. [21], who first applied the idea of code smells to test code and proposed corresponding refactorings, and has since been widely studied and tooled [1, 8, 20].

Most of this body of work, however, targets functional unit-test smells. Performance tests have their own recurring bad practices, for instance, running costly operations in the per-virtual-user initialization context, omitting thresholds, which turns the test into a pass-by-default check, not validating responses, or failing to tag requests for later analysis. Samuel Barbosa [3] formalized four such performance test smells for k6 and built a static linter and an annotated dataset to detect them. These smells are particularly insidious because, unlike many functional smells, they can silently produce false confidence: a test may report success even when the system under test is underperforming.

In parallel, Large Language Models (LLMs) have emerged as a promising, low-setup alternative to static analyzers for code-quality tasks. Recent work has shown that LLMs can detect a range of conventional test smells [11] and can recommend code refactorings whose quality can rival that of human experts in certain settings when adequately prompted [5, 16]. Two gaps motivate this study. First, prior evaluations focus almost exclusively on functional test

smells in languages such as Java [1, 11], leaving performance test smells unexamined. Second, and more fundamentally, detection and refactoring are usually treated as a single capability, even though correctly flagging a smell offers no guarantee that a model can correctly fix it. Whether LLMs can reliably detect and repair performance test smells, and how prompt design affects this, remains an open question.

This paper addresses that question through an empirical study of three widely used LLMs (Claude, GPT, and Gemini) on a dataset of 228 real-world k6 scripts drawn from 58 GitHub repositories and labeled by the `k6-performance` linter [3]. Each script was submitted to each model under two prompt variants (definitions only, and definitions plus illustrative code examples), yielding 1,368 detection-and-refactoring queries, complemented by a manual review of 120 refactoring suggestions by a domain expert. We assess detection with standard classification metrics and refactoring quality through a four-level qualitative rubric.

Our results yield three main findings. First, detection performance is moderate and broadly comparable across the three models, but all of them struggle with smells defined by the absence of a configuration or behavioral element. Second, contrary to our expectation, augmenting the prompt with worked code examples does not consistently improve detection and, for two of the three models, degrades it. Third, and most notably, detection difficulty and refactoring difficulty are largely decoupled: the smells that are hardest to detect are refactored almost flawlessly once identified, whereas one of the best-detected smells concentrates nearly all of the unsafe refactorings we observed. We trace this asymmetry to a distinction between declarative fixes (inserting configuration, a validation call, or metadata) and architectural fixes (reasoning about k6's execution lifecycle), and argue that it has direct consequences for how much human review an LLM-suggested refactoring warrants.

In summary, this paper makes the following contributions:

- an empirical evaluation of three state-of-the-art LLMs on the detection of four performance test smells in 228 real-world k6 scripts, under two prompt designs;
- a manual, expert-based assessment of the *quality* of the refactorings these models produce;
- evidence that detection and refactoring are decoupled capabilities, together with a declarative-versus-architectural distinction that helps explain when LLM-suggested fixes can be trusted.

2 Performance Test Smells

This study focuses on four performance-related test smells commonly found in k6 scripts. The k6 framework's official documentation [6] outlines several best practices to help avoid these smells; however, the formalization of performance test smells is derived from the work developed by Barbosa [3].

This section presents the four smells addressed in this study: Costly operations in the init context (S1), Do not use thresholds (S2), Not checking requests (S3) and Not using tags (S4). The following subsections provide a detailed description of each smell.

2.1 Costly Operations in the Init Context (S1)

Description: The init context runs once per virtual user before the test starts. Performing expensive operations in this phase, such as reading large files, parsing JSON, or executing heavy computations, can degrade performance and increase resource usage unnecessarily.

How To Refactor: Move one-time logic to setup() and move data shared across VUs (Virtual Users) into a SharedArray, keeping the init context free of file reads, parsing, or heavy computation.

Original

```
import { open } from "k6";
const largeDataFile = open("./large-data.json");
const testData = JSON.parse(largeDataFile);
for (let i = 0; i < 100; i++) {
    console.log(i);
}
```

Refactored

```
import { open } from "k6";
import { SharedArray } from "k6/data";
const data = new SharedArray("test", function() {
    const largeDataFile = open("./large-data.json");
    return JSON.parse(largeDataFile);
});
export default function setup() {
    for (let i = 0; i < 100; i++) {
        console.log(i);
    }
}
```

Figure 1: Example of Refactoring Smell 1.

2.2 Do Not Use Thresholds (S2)

Description: Thresholds define success or failure conditions based on test metrics. Without them, a test may always be reported as successful even when the system is underperforming, creating false confidence, especially in CI/CD pipelines.

How To Refactor: Add a thresholds block in options, defining pass/-fail conditions on built-in or custom metrics so the test result reflects actual system performance.

Original

```
import http from 'k6/http';
export const options = {
vus: 50,
duration: '1m',
};
export default function () {
http.get('https://api.minha-aplicacao.com');
}
```

Refactored

```
import http from 'k6/http';
export const options = {
  vus: 50,
  duration: '1m',
  thresholds: {
    'http_req_failed': ['rate<0.01'],
    'http_req_duration': ['p(95)<200'],

  },
};
export default function () {
  http.get('https://api.minha-aplicacao.com');
}
```

Figure 2: Example of Refactoring Smell 2.

2.3 Not Checking Requests (S3)

Description: k6 does not fail a test when HTTP requests return error responses (4xx or 5xx). Instead, it records the failure metrics and continues execution. Without explicit response validations, such as checks on status codes or response content, tests may incorrectly pass even when the system is returning errors, leading to false-positive results.

How To Refactor: Use check() to validate response correctness (status, body, etc.) and add a threshold on the checks metric so invalid responses fail the test.

Original

```
import http from 'k6/http';
import { sleep } from 'k6';
export const options = {
thresholds: {
http_req_duration: ['p(95)<500'],
},
};
export default function () {
http.get('https://api.minha-aplicacao.com/dados-criticos');
sleep(1);
}
```

Refactored

```
import http from 'k6/http';
import { check, sleep } from 'k6';
export const options = {
thresholds: {
http_req_duration: ['p(95)<500'],
'checks': ['rate>0.99'],
},
};
export default function () {
const res = http.get('https://api.minha-aplicacao.com/dados-criticos
');
check(res, {
'status is 200': (r) => r.status === 200,
});
sleep(1);
}
```

Figure 3: Example of Refactoring Smell 3.

2.4 Not Using Tags (S4)

Description: Tags are key-value metadata that can be attached to requests, metrics, and checks in k6. Without tags, it becomes difficult to filter and group metrics by endpoint or user flow, making it harder to pinpoint the source of performance issues.

How To Refactor: Add tags to requests/checks/metrics to label each step, enabling per-tag thresholds and result filtering to pinpoint bottlenecks.

Original

```
import http from 'k6/http';
import { check } from 'k6';
export const options = {
thresholds: {
'http_req_duration{name:Login}': ['p(95)<300'],
'http_req_duration{name:Products}': ['p(95)<1000'],
'http_req_duration{name:Cart}': ['p(95)<500'],
},
};
```

Refactored

```
import http from 'k6/http';
import { check } from 'k6';
export const options = {
thresholds: {
'http_req_duration{name:Login}': ['p(95)<300'],
'http_req_duration{name:Products}': ['p(95)<1000'],
'http_req_duration{name:Cart}': ['p(95)<500'],
},
};
export default function () {
let resLogin = http.get('https://api.minha-loja.com/v2/login', {
tags: { name: 'Login' }
});
let resProducts = http.get('https://api.minha-loja.com/v2/products',
{
tags: { name: 'Products' }
});
let resCart = http.get('https://api.minha-loja.com/v2/cart', {
tags: { name: 'Cart' }
});
}
```

Figure 4: Example of Refactoring Smell 4.

3 Research Methodology

This study aims to assess the extent to which modern Large Language Models (LLMs) are capable of detecting and refactoring performance test smells in load testing scripts written using the k6 framework. This study was inspired by the work of Lucas et al. [9], which explored the potential of Small Language Models (SLMs) for the automatic detection of test smells. Given the significance and promising results of that research, we adopted its methodological approach as the foundation for the present study. The investigation is guided by four research questions:

- **RQ1**: Can LLMs detect and refactor performance smells in k6 scripts when prompted only with the smell definitions?
- **RQ2**: Can LLMs detect and refactor performance smells in k6 scripts when prompted with the smell definitions accompanied by illustrative code examples?
- **RQ3**: How does providing illustrative code examples affect LLM performance compared to providing definitions alone?
- **RQ4**: Among the three most widely adopted LLMs at the time of this study, which one achieves the highest detection accuracy?

3.1 Model Selection

The selection of the models GPT (OpenAI), Claude (Anthropic), and Gemini (Google DeepMind) was based on the fact that they represent the leading families of proprietary Large Language Models currently available and widely recognized as the state of the art in Generative Artificial Intelligence [4]. These models present high scientific and industrial relevance, and cover different dimensions of performance, including logical reasoning, multimodal understanding, large-context processing, and natural language generation. The choice of these three families allows for a comprehensive comparison between leading market approaches, reducing biases arising from the analysis of a single provider and increasing the external validity of the obtained results.

The specific model versions evaluated via their official APIs were:

Table 1: LLMs evaluated in this study.

Provider	LLM	Model Version
Anthropic	Claude	`claude-sonnet-4-6`
OpenAI	GPT	`gpt-5.5-2026-04-23`
Google DeepMind	Gemini	`gemini-2.5-pro`

3.2 Dataset Selection

The dataset used in this study comprises 228 real-world k6 load testing scripts, extracted from open-source repositories hosted on GitHub. The collection and annotation of these scripts were performed by the `k6-performance` linter developed by Barbosa [3], a static analysis tool designed to automatically detect performance-related test smells in k6 scripts. For each analyzed script, the linter produces a structured annotation indicating the presence and exact location (line and column) of each of the four smell types (S1-S4), which constitutes the ground-truth label used for evaluating the LLMs' responses in this study.

The raw output produced by the linter required two data cleaning phases prior to its use in the experiment. In the first phase, the initial set of 428 candidate repositories, containing a total of 3,073 scripts, was filtered to retain only those that were publicly accessible and that contained at least one k6 script with a smell annotated by the linter, reducing the pool to 212 valid repositories and 2,105 scripts. In the second phase, a simple random sample was drawn from these 212 repositories, yielding a final set of 58 repositories and a total of 228 scripts. This reduction was motivated by practical constraints related to the token consumption of the LLM APIs selected for this study: submitting the full corpus of 2,105

scripts would have incurred prohibitive API costs while yielding diminishing methodological returns, given that the sampled set preserves the distribution of smell types present in the full corpus.

The resulting dataset of 228 scripts, each labeled with the set of smells detected by the linter, serves as the ground truth against which the LLMs' responses are automatically compared in the quantitative evaluation described in Section 3.4.

Table 2 reports the prevalence of each smell in this ground truth, that is, the number of scripts labeled positive for each smell type. The dataset is markedly imbalanced across the four categories: S1 is present in only 15 of the 228 scripts (6.6%), whereas S4 is present in 146 (64.0%). This distribution is relevant for interpreting the results reported in Section 4, since micro-averaged metrics are dominated by the most frequent smells, whereas macro-averaged metrics give equal weight to rare and frequent ones.

Table 2: Prevalence of each smell in the ground truth (228 scripts).

Smell	Positive scripts	% of dataset
S1 - Costly init	15	6.6%
S2 - No thresholds	56	24.6%
S3 - No checks	87	38.2%
S4 - No tags	146	64.0%

3.3 Prompt Design

Two prompt variants were designed, corresponding to RQ1 and RQ2, respectively. Both prompts position the model as an expert in performance engineering, load testing, and k6 scripting, and present the four smells with their identifying numbers, names, and conceptual definitions [16]:

- **Prompt 1 - Description-only (RQ1)**: presents only the textual definition of each of the four smells, followed by the k6 script under analysis and an instruction to identify any present smells, point out their exact location, and suggest a refactored version (Figure 5).
- **Prompt 2 - Description + Example (RQ2)**: extends Prompt 1 by adding, for each smell, a concrete illustrative code snippet demonstrating the smell (Figure 6).

```
You are an expert in performance engineering, load testing, and k6
scripting.
Review the following k6 code and identify whether it contains any of the
four performance smells described below. For each smell, a number (1–4), a
name, and a definition are provided.
SMELL 1 — Costly operations in the init context
The init context runs once per virtual user before the test starts.
Performing expensive operations in this phase — such as reading large files
, parsing JSON, or executing heavy computations — can degrade performance
and increase resource usage unnecessarily.
SMELL 2 — Do not use thresholds
Thresholds define success or failure conditions based on test metrics.
Without them, a test may always be reported as successful even when the
system is underperforming, creating false confidence — especially in CI/CD
pipelines.
SMELL 3 — Not checking requests
k6 does not automatically fail a test when HTTP requests return 4xx or 5xx
responses; it only updates failure-related metrics and continues execution.
 Without explicit checks validating response correctness, the script may
produce false positives and incorrectly report successful results.
SMELL 4 — Not using tags
Tags are key-value metadata that can be attached to requests, metrics, and
checks in k6. Without tags, it becomes difficult to filter and group
metrics by endpoint or user flow, making it harder to pinpoint the source
of performance issues.
TASK:
Analyze the k6 code provided below and determine whether it contains one or
 more of the smells above (1, 2, 3, or 4).
For each smell detected:
- Identify which smell it is (number and name).
- Point out exactly where in the code it occurs.
- Suggest a refactored version for code with a code smell.
If no smell is found, state that the code does not exhibit any of the four
performance smells described.
```

Figure 5: Prompt 1 - Description-only.

```
You are an expert in performance engineering, load testing, and k6
scripting.
Review the following k6 code and identify whether it contains any of the
four performance smells described below. For each smell, a number (1–4), a
name, a definition, and a concrete example are provided.
SMELL 1 — Costly operations in the init context
Definition:
The init context runs once per virtual user before the test starts.
Performing expensive operations in this phase — such as reading large files
, parsing JSON, or executing heavy computations — can degrade performance
and increase resource usage unnecessarily.
Example:
import { open } from "k6";
const largeDataFile = open("./large-data.json");
const testData = JSON.parse(largeDataFile);       // heavy parsing in init
for (let i = 0; i < 100; i++) {
  console.log(i);                                  // unnecessary loop in
init
}
SMELL 2 — Do not use thresholds
Definition:
Thresholds define success or failure conditions based on test metrics.
Without them, a test may always be reported as successful even when the
system is underperforming, creating false confidence — especially in CI/CD
pipelines.
Example:
import http from 'k6/http';
export const options = {
  vus: 50,
  duration: '1m',
  // no thresholds defined
};
export default function () {
  http.get('https://api.minha-aplicacao.com');
}
SMELL 3 — Not checking requests
Definition:
k6 does not automatically fail a test when HTTP requests return 4xx or 5xx
responses; it only updates failure-related metrics and continues execution.
 Without explicit checks validating response correctness, the script may
produce false positives and incorrectly report successful results.
Example:
import http from 'k6/http';
import { sleep } from 'k6';
export const options = {
  thresholds: {
    http_req_duration: ['p(95)<500'],
  },
};
export default function () {
  http.get('https://api.minha-aplicacao.com/dados-criticos');  // no check
()
  sleep(1);
}
SMELL 4 — Not using tags
Definition:
```

```
Tags are key-value metadata that can be attached to requests, metrics, and
checks in k6. Without tags, it becomes difficult to filter and group
metrics by endpoint or user flow, making it harder to pinpoint the source
of performance issues.
Example:
import http from 'k6/http';
import { check } from 'k6';
export const options = {
  thresholds: {
    'http_req_duration{name:Login}': ['p(95)<300'],
    'http_req_duration{name:Products}': ['p(95)<1000'],
    'http_req_duration{name:Cart}': ['p(95)<500'],
  },
};
export default function () {
  http.get('https://api.minha-aplicacao.com/login');     // no tags
attached
  http.get('https://api.minha-aplicacao.com/products');  // no tags
attached
  http.get('https://api.minha-aplicacao.com/cart');      // no tags
attached
}
TASK:
Analyze the k6 code provided below and determine whether it contains one or
 more of the smells above (1, 2, 3, or 4).
For each smell detected:
- Identify which smell it is (number and name).
- Point out exactly where in the code it occurs.
- Suggest a refactored version for code with a code smell.
If no smell is found, state that the code does not exhibit any of the four
performance smells described.
```

Figure 6: Prompt 2 - Description + Example.

Both prompts request, for each detected smell, (i) its identifier and name, (ii) the exact location in the code, and (iii) a refactored version of the affected snippet. If no smell is detected, the model is instructed to explicitly state this.

3.4 Test Smell Detection Workflow

The overall workflow adopted for performance test smell detection and evaluation is illustrated in Figure 7. Each of the 228 k6 scripts in the dataset was submitted to each of the three LLMs under both prompt variants, totaling $228 \times 3 \times 2 = 1{,}368$ individual queries. Each query consisted of a single API call, with temperature and other generation parameters kept at their default values, without resampling. For each response, the smells reported by the LLM were automatically extracted and compared against the ground-truth labels from the dataset.

For each script, the set of smells reported by the LLM was compared against the ground-truth labels using a binary presence/absence classification for each of the four smell types (S1–S4). Based on this comparison, the following metrics were computed:

- **Precision, Recall, and F1-score (micro-averaged)**: computed by pooling true positives, false positives, and false negatives across all smell types and all scripts, providing an overall measure of detection performance.
- **F1-score (macro-averaged)**: computed by first calculating the F1-score independently for each smell type (S1-S4) and then averaging these four values, giving equal weight to each smell category regardless of its frequency in the dataset.
- **Accuracy**: the average binary classification accuracy (correct presence/absence predictions) across the four smell types, per script, averaged over the dataset.
- **Per-smell F1-score**: the F1-score computed individually for each of the four smell types (S1, S2, S3, S4), enabling a finer-grained comparison of which smells are easier or harder for each LLM to detect.

In addition to these automatic detection metrics, the quality of the refactoring suggestions produced by the LLMs was assessed manually by a domain expert with practical experience in performance testing with k6. Given the high cost of reviewing all 1,368 generated responses, a stratified sample was drawn, restricted to cases in which the corresponding LLM correctly detected the smell under evaluation. For each combination of LLM, prompt variant (P1, P2), and smell type (S1-S4), five cases were randomly sampled, yielding exactly 120 refactoring suggestions.

Each sampled refactoring was classified into one of four categories:

- **Correct**: the suggested refactoring fully addresses the identified smell, preserves the original semantics and intent of the test script, and follows idiomatic k6 best practices.
- **Partial**: the refactoring addresses the smell only partially or introduces minor imprecisions that do not compromise script execution.
- **Incorrect**: the refactoring does not address the identified smell, addresses a different smell than the one reported, or reflects a conceptual misunderstanding of the smell.
- **Breaking**: the refactoring introduces syntax errors, structural inconsistencies, or semantic changes that would prevent the script from executing correctly or would alter its intended test behavior.

The results are presented and discussed in detail in Section 4.

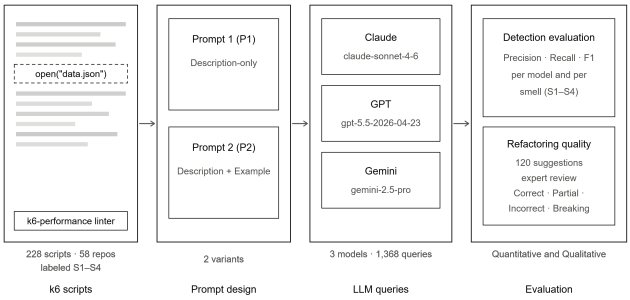

Figure 7: Overview of the study design.

4 Results

This section presents the results obtained from the 1,368 queries submitted to the three evaluated LLMs (Claude, GPT, and Gemini) under both prompt variants (P1: description-only and P2: description with illustrative examples), as described in Section 3. The analysis is organized into two parts: a quantitative analysis (Section 4.1), based on the automatic detection metrics computed against the ground-truth labels of the dataset, and a qualitative analysis (Section 4.2), based on the manual evaluation of the refactoring suggestions produced by the LLMs.

4.1 Quantitative Analysis

In this study, the F1 metric was adopted as the primary evaluation criterion, as it represents the harmonic mean of precision and recall, providing a balanced measure of model performance in binary classification scenarios. In code smell detection tasks, this choice is particularly relevant given that the dataset is inherently imbalanced, that is, the number of positive instances (snippets exhibiting the smell) tends to be considerably smaller than that of negative instances. In this context, metrics such as accuracy can be misleading, since a model that systematically predicts the majority class would still achieve high values without actually detecting any smell.

The F1 score, by equally penalizing false positives and false negatives, more faithfully reflects the model's actual ability to identify the problematic patterns of interest. Both micro F1, which aggregates the contributions of all instances weighted by class frequency, and macro F1, which computes the unweighted arithmetic mean of per-class scores and thus assigns equal importance to rare and frequent smells, are reported.

Table 3 summarizes the overall detection performance of the three LLMs under both prompt variants, while Table 4 breaks down the F1-score by smell type.

Table 3: Overall detection metrics by LLM and prompt variant.

LLM	Prec.	Recall	F1 (micro)	F1 (macro)	Acc.
Prompt 1 - Description-only:					
Gemini	0.4428	0.7007	0.5427	0.5544	0.6064
Claude	0.4430	0.6513	0.5273	0.5222	0.6107
GPT	0.4584	0.6349	0.5324	0.5218	0.6283
Prompt 2 - Description + Example:					
Gemini	0.4417	0.6480	0.5253	0.5628	0.6096
Claude	0.4344	0.5559	0.4877	0.5007	0.6107
GPT	0.4571	0.5954	0.5171	0.4904	0.6294

Table 4: Per-smell F1-score by LLM and prompt variant.

	S1		S2		S3		S4	
LLM	P1	P2	P1	P2	P1	P2	P1	P2
Gemini	0.7568	0.8571	0.3628	0.3687	0.3626	0.3218	0.7353	0.7037
Claude	0.7059	0.7273	0.3704	0.3902	0.2738	0.2208	0.7387	0.6645
GPT	0.6452	0.5714	0.3805	0.3838	0.3537	0.3038	0.7077	0.7025

4.1.1 RQ1 - Detection and Refactoring Without Code Examples (Prompt 1). Under the description-only prompt, all three LLMs achieved comparable overall performance, with F1 micro scores ranging from 0.5273 (Claude) to 0.5427 (Gemini) and accuracy values between 0.6064 and 0.6283. Gemini obtained the highest F1 macro (0.5544), driven primarily by strong performance on S1 (Costly operations in the init context, F1 = 0.7568) and S4 (Not using tags, F1 = 0.7353). Across all three models, S2 (Do not use thresholds) and S3 (Not checking requests) consistently presented the lowest F1 scores, generally between 0.2738 and 0.3805, indicating that LLMs are markedly less effective at identifying these two smells from definitional prompts alone. These results suggest that, even without access to illustrative code, current LLMs are capable of producing a useful, though imperfect, first-pass detection of structural smells such as S1 and S4, while smells that depend on absent configuration or behavioral elements (S2, S3) remain considerably harder to detect.

4.1.2 RQ2 - Detection and Refactoring With Code Examples (Prompt 2). When the prompt was extended with concrete code examples for each smell, the overall picture changed in a counterintuitive direction. F1 micro scores decreased for all three models: the drop was substantial for Claude (0.4877, down from 0.5273, $\Delta = -0.0396$) and small for GPT (0.5171 vs. 0.5324, $\Delta = -0.0153$) and Gemini (0.5253 vs. 0.5427, $\Delta = -0.0174$). Gemini was the only model whose F1 macro improved with the addition of examples (0.5628 vs. 0.5544, $\Delta = +0.0084$), driven by a substantial gain on S1 (F1 = 0.8571, $\Delta = +0.1003$). However, this gain was accompanied by losses on S3 ($\Delta = -0.0408$) and S4 ($\Delta = -0.0316$), suggesting that the example for S1 dominated the model's attention at the expense of the other categories. A plausible explanation for this behavior is an attentional anchoring effect: the presence of a concrete, structurally salient code example for S1, featuring a file read, a JSON parse, and a loop in the init context, may have shifted the models from abstract reasoning over smell definitions toward surface-level pattern matching against that specific snippet, reducing their sensitivity to the remaining smell categories. This effect is particularly consequential for S2 and S3, which are absence-based smells that manifest precisely when a construct is missing from the code; a worked example of what the smell looks like offers limited guidance for detecting an omission, and may in fact reinforce the tendency to search for a visible pattern rather than notice its absence.

4.1.3 RQ3 - Impact of Illustrative Code Examples on Performance. Averaging across the three LLMs, Prompt 1 (description-only) outperformed Prompt 2 (description with examples) on most aggregate metrics, particularly recall. GPT showed the largest overall F1 macro decline ($\Delta = -0.0314$), followed by Claude ($\Delta = -0.0215$). The two models, however, declined on different smells: for GPT the largest per-smell drops were on S1 ($\Delta = -0.0738$) and S3 ($\Delta = -0.0499$), with S4 essentially unchanged ($\Delta = -0.0052$), whereas for Claude they were on S4 ($\Delta = -0.0742$) and S3 ($\Delta = -0.0530$), with S1 slightly improving ($\Delta = +0.0214$). Only Gemini benefited from the additional examples in terms of F1 macro, and even then the benefit was concentrated almost entirely on S1, at the cost of S3 and S4. A plausible explanation is an attentional or anchoring effect: the presence of a fully worked example for one smell (S1) may bias the model toward pattern-matching against that specific example, reducing its sensitivity to the other three smell categories, whose corresponding examples, despite being present in the prompt, appear to receive comparatively less weight in the model's reasoning.

4.1.4 RQ4 - Comparative Accuracy Among the Three LLMs. Considering Prompt 1 (the better-performing configuration overall for two of the three models), Gemini achieved the highest F1 micro (0.5427) and F1 macro (0.5544), followed in F1 micro by GPT (0.5324) and Claude (0.5273); in F1 macro, Claude (0.5222) and GPT (0.5218) are effectively tied, since the difference between them is only 0.0004. In terms of accuracy alone, however, GPT obtained the highest value (0.6283), followed closely by Claude (0.6107) and Gemini (0.6064), an apparent inversion explained by the fact that accuracy reflects the average per-smell binary classification rate, which is less sensitive to recall on rare positive cases (such as S1, present in only 15 of the

228 scripts, as reported in Table 2) than the F1-based metrics. Under Prompt 2, Gemini's lead widened in F1 macro (0.5628 vs. 0.5007 for Claude and 0.4904 for GPT), while GPT retained the highest accuracy (0.6294). Taken together, these results indicate that Gemini provided the most balanced detection performance across the four smell categories under both prompting conditions, particularly excelling at S1, with S4 performance on par with the other models, whereas Claude and GPT showed comparatively higher per-smell accuracy but lower macro-averaged F1, reflecting a tendency toward more conservative (lower-recall) predictions, especially for S2 and S3.

4.2 Qualitative Analysis

In addition to the automatic detection metrics reported above, the quality of the refactoring suggestions produced by the LLMs was assessed manually by a domain expert with practical experience in performance testing with k6. Given the prohibitively high cost of reviewing all 1,368 generated responses, a stratified sample was drawn, restricted to cases in which the corresponding LLM correctly detected the smell under evaluation. For each combination of LLM, prompt variant (P1, P2), and smell type (S1-S4), five cases were randomly sampled, yielding exactly $3 \times 2 \times 4 \times 5 = 120$ refactoring suggestions.

Each sampled refactoring was classified into one of four categories: Correct, Partial, Incorrect and Breaking. Each category is described in Section 3.4.

Table 5: Overall distribution of refactoring quality across all reviewed suggestions.

Category	Count	%
Correct	94	78.3%
Partial	18	15.0%
Incorrect	5	4.2%
Breaking	3	2.5%
Total	120	100%

Table 6: Refactoring quality by model and prompt variant.

Model	Correct	Partial	Incorrect	Breaking	Total
Claude (P1)	17	2	1	0	20
Claude (P2)	14	2	3	1	20
GPT (P1)	16	4	0	0	20
GPT (P2)	17	3	0	0	20
Gemini (P1)	15	4	0	1	20
Gemini (P2)	15	3	1	1	20

Table 7: Refactoring quality by smell type (S1-S4).

Smell	Correct	Partial	Incorrect	Breaking	Total
S1 - Costly init	13	10	5	2	30
S2 - No thresholds	29	1	0	0	30
S3 - No checks	28	2	0	0	30
S4 - No tags	24	5	0	1	30

Illustrative Examples. To provide concrete intuition for each quality category, we present one representative example of each of the Correct, Partial, and Breaking categories.

Original

```
export default function () {
    homeWCAdmin();
}

```

LLM Suggestion

```
export const options = {
    thresholds: {
        'http_req_duration{name:
       Merchant - WC-Admin}': ['p(95)
       <500'],
        'http_req_duration{name:
       Merchant - wc-admin/onboarding/
       tasks?}': ['p(95)<500'],
        'http_req_duration{name:
       Merchant - wc-analytics/admin/
       notes?}': ['p(95)<500'],
        'http_req_failed': ['rate
       <0.01'],
    },
};
export default function () {
    homeWCAdmin();
}

```

Figure 8: Example 1 - Correct refactoring (Claude, P2, S2).

The script originally defined no thresholds at all. Claude introduces a complete and well-scoped `thresholds` block, with one entry per named endpoint plus a global `http_req_failed` guard, fully addressing S2 while leaving the original test logic unchanged.

Original

```
const response =
       concreteProductsResource.get(
       product.sku);
metrics[testConfiguration.metrics[0]].
       add(response.timings.duration);

```

LLM Suggestion

```
export default function (data) {
  const product = fixture.iterateData(
       data);
  const tags = {
    api: testConfiguration.id,
    endpoint: 'get_concrete_products',
    flow: testConfiguration.group,
  };
  group(testConfiguration.group, () =>
       {
    const concreteProductsResource =
       new ConcreteProductsResource();
    const response =
       concreteProductsResource.get(
       product.sku, { tags });
    metrics[testConfiguration.metrics
       [0]].add(response.timings.
       duration, tags);
  });
}

```

Figure 9: Example 2 - Partial refactoring (GPT, P1, S4).

GPT correctly tags the custom metric call (`.add(value, tags)`, a documented k6 API) and constructs a sensible tag object. However, the `tags` parameter is also forwarded as a second argument to `concreteProductsResource.get()`, a custom page-object/resource wrapper whose implementation is not shown; nothing in the visible code confirms that this wrapper actually propagates the `tags` option down to the underlying `http` call. The fix is plausible, but HTTP-level tagging remains unverified, which is exactly the imprecision that justifies a Partial rating rather than Correct.

Original

```
const db = sql.open('postgres',
       pdConnectionString)

```

LLM Suggestion

```
export function setup() {
  const db = sql.open('postgres',
        pdConnectionString);
  return { db: db };
}
export default function(data) {
  // use data.db instead of the global
        db
}
export function teardown(data) {
  data.db.close();
}

```

Figure 10: Example 3 - Breaking refactoring (Gemini, P1, S1).

Gemini correctly diagnoses that the connection should be created once rather than per VU, and moves it into `setup()`. However, the value returned by `setup()` is serialized to JSON before being distributed to each VU and to `teardown()`; a live database connection object cannot survive this serialization. As written, both `data.db` inside the default function and `data.db.close()` inside `teardown()` would fail at runtime, since the connection handle is lost in transit. This reflects a structural misunderstanding of k6's execution model that would prevent the script from running correctly, warranting a Breaking classification.

Discussion of Refactoring Quality. Across the 120 reviewed refactorings, 78.3% were classified as fully Correct, and a further 15.0% as Partial, meaning that more than 93% of the sampled suggestions provided a usable starting point for fixing the corresponding smell. Outright Incorrect or Breaking refactorings were comparatively rare (4.2% and 2.5%, respectively), but were not evenly distributed across smell types or models.

At the model level, GPT showed the most consistent behavior, with zero Incorrect or Breaking refactorings across both prompt variants. In P1, Gemini produced the single Breaking case and Claude the single Incorrect case observed under that prompt variant. In P2, Claude produced three Incorrect refactorings and one Breaking case, and Gemini produced one Incorrect and one Breaking case. All of Claude's Incorrect refactorings were concentrated in S1, where it repeatedly acknowledged the smell in a comment but left the costly parsing in the per-VU init scope unchanged. No clear, consistent advantage of one prompt variant (P1 vs P2) over the other emerged at the aggregate level; differences between P1 and P2 for the same model were generally within one or two cases out of 20.

5 Discussion & Further Analysis

The quantitative and qualitative analyses presented in Section 4, when examined separately, paint two seemingly inconsistent pictures of how well the evaluated LLMs handle performance test smells in k6 scripts. Bringing them together reveals a pattern that neither analysis exposes on its own.

Across all three LLMs and both prompt variants, S2 (missing thresholds) and S3 (missing checks) were consistently the hardest smells to *detect*, with F1 scores ranging from 0.2208 to 0.3902 (Table 4). Yet, in the qualitative sample, these two smells were refactored almost flawlessly once detected: 29 of 30 sampled S2 cases and 28 of 30 sampled S3 cases were rated Correct (Table 7), with zero Incorrect or Breaking outcomes for either category.

S1 (costly operations in the init context) shows the opposite profile. Together with S4, it was among the most reliably *detected* smells: its per-smell F1 ranged from 0.5714 to 0.8571, and its average across all LLM and prompt combinations (0.7106) was far above those of S2 (0.376) and S3 (0.306), essentially on par with S4 (0.7087), the other well-detected smell. However, S1 was by far the hardest to *refactor* correctly: only 13 of 30 sampled cases (43.3%) were rated Correct, and S1 alone accounted for all 5 Incorrect and 2 of the 3 Breaking refactorings observed across the entire 120-case sample. This inversion is the central finding of this discussion: detection performance and refactoring performance are not two views of the same underlying capability, but largely independent dimensions of LLM competence. A model that reliably flags a smell provides no guarantee that it can reliably fix it, and a model that struggles to flag a smell may, paradoxically, fix it almost perfectly once the smell is pointed out. Studies that evaluate only detection accuracy therefore capture only half of the risk surface relevant to using LLMs as refactoring assistants.

5.1 Implications for Trusting LLM-Generated Refactorings

Taken together, these observations indicate that the reliability of an LLM-suggested refactoring is not a property of the smell category in isolation, but of the *type of change* the fix requires. Declarative fixes, inserting configuration, adding a standard validation call, or attaching metadata, were the most reliable in our sample: for S2 and S3 alone, 57 of 60 sampled refactorings (95%) were rated Correct, with zero Incorrect or Breaking outcomes (the remaining three rated Partial), and across all non-S1 (declarative) smells the Correct rate was 90% (81 of 90), with errors confined almost entirely to coverage gaps (e.g., tagging only a subset of flagged call sites in S4) rather than structural mistakes. Architectural fixes—those requiring reasoning about execution order, scope, or data lifetime across the test's lifecycle—exhibited substantially lower reliability and accounted for every Incorrect refactoring and all but one Breaking refactoring observed in the study.

This asymmetry suggests that blanket trust policies for LLM-assisted test refactoring (e.g., "accept all suggestions" or "reject all suggestions") are both suboptimal: the former exposes the codebase to a meaningful risk of non-executable scripts concentrated in exactly the smells whose detection looks most reliable from F1 scores alone, while the latter discards a class of fixes that, in our sample, were essentially always correct. A more calibrated approach would condition the level of required human review on whether the underlying fix is declarative or architectural in nature, a distinction that, notably, is not recoverable from the smell's detection F1 score and must instead be assessed per smell type based on the nature of its remediation.

5.2 Threats to Validity

Some threats to validity should be considered when interpreting the results of this study. First, Large Language Models are subject to frequent updates, and newer versions or alternative models may yield different results when applied to the same task. Consequently, replicating this experiment with other LLMs could lead to variations in the detection of performance-related test smells.

Second, the qualitative analysis was conducted by a single reviewer, which may introduce bias in the interpretation of the results; future work should involve an additional expert reviewer to improve the reliability and accuracy of this analysis. Finally, the ground-truth labels used in this study were produced by the k6-performance static linter developed by Barbosa [3], which is itself subject to known construct validity limitations. As Barbosa acknowledges, the linter is based on AST-pattern matching and may produce false negatives, for instance, when tags are generated dynamically or checks are encapsulated in auxiliary functions that the rule does not cover, as well as false positives in contexts where the flagged pattern is structurally similar to a smell but contextually justified, such as the intentional absence of thresholds in smoke or debugging scripts.

Our study treats the linter's output as a reasonable and reproducible approximation of the ground truth rather than as a definitive oracle, consistent with the intended scope of the tool as described by its author. Selecting a different random sample from the 212 valid repositories, or applying a different linter configuration, could lead to variations in the evaluation results.

Additionally, the scope of this study is restricted to four performance test smells, as defined by Barbosa [3]. This limited catalog may not reflect the full range of bad practices present in real-world k6 scripts, and conclusions drawn from these four smells may not generalize to other smell types. Studies considering a broader set of smells could reveal different patterns of LLM behavior in both detection and refactoring.

6 Related Work

Samuel Barbosa [3] proposed the `k6-performance` linter and an annotated dataset of real-world k6 scripts covering four types of performance test smells. This work is the direct foundation of our study: both the smell catalog (S1-S4) and the ground-truth dataset of 228 k6 scripts used in our experiment are derived from the tool and dataset made publicly available by Barbosa.

Lucas et al. [9] used Small Language Models (Gemma3, Llama3.2, and Phi-4) with Directional Stimulus Prompting to detect test smells in manual test cases written in natural language, with Phi-4 achieving the best results. More relevant to our study, the models suggested refactorings autonomously even without explicit prompt instructions, indicating that LLMs can go beyond detection, a capability we assess directly on k6 scripts.

Extending this line of research to a contemporary large model, Lucas et al. [10] evaluated Gemini-3 on whole manual test cases from the Ubuntu repository, reaching 0.92 precision and 0.91 recall and outperforming the previously evaluated SLMs while producing actionable, evidence-grounded explanations. Notably, the authors operationalize only detection and explicitly leave the refactoring of the smelly tests as future work, reinforcing that, even for the strongest models reported, refactoring quality remains unmeasured, a gap our study addresses for performance smells in k6.

Soares et al. [19] consolidated a catalog of 480 test smells and proposed refactoring transformations validated through 38 pull requests to open-source projects, with a 94% acceptance rate. This result shows that developers are receptive to automated refactoring suggestions when these are well-targeted, but it relies on transformations manually engineered per smell type and per framework feature. Our study investigates whether general-purpose LLMs can produce comparably acceptable refactoring suggestions for performance test smells in k6 without hand-crafted, smell-specific transformation rules.

In the same spirit of hand-crafted refactoring, Aranda et al. [2] proposed a catalog of seven transformations to remove natural-language test smells, implemented in an NLP-based tool and validated by 15 testing professionals with a 91.43% acceptance rate, while the tool itself reached an 83.70% F-measure. This finding further suggests that developers value automated refactoring suggestions, but, like Soares et al., the approach depends on transformations crafted manually per smell type rather than generated by a general-purpose model.

Lucas et al. [11] evaluated three general-purpose LLMs (ChatGPT-4, Gemini Advanced, and Mistral Large) on the detection of 30 test smell types across seven programming languages under a single zero-shot prompt, with ChatGPT-4 performing best. Their evaluation was restricted to detection over small curated snippets and explicitly left refactoring to future work, a gap we address on real-world k6 scripts and across two prompt designs.

Aljedaani et al. [1] cataloged 22 test smell detection tools and found that the majority target Java and the JUnit framework, that only five offer refactoring support, and that none report the accuracy of the refactorings they generate. This twofold gap, narrow language coverage and unmeasured refactoring quality, reinforces the relevance of investigating performance test smells in k6 and of evaluating the correctness of the fixes, not merely their detection.

Rule- and NLP-based detection was the dominant paradigm before the emergence of LLMs. Soares et al. [17] cataloged eight natural-language test smells and built an NLP-based detector that reached a 93.5% F-measure over real-world manual tests, illustrating both the effectiveness and the manual rule-engineering cost of this approach. As an alternative to such heuristics, Pontillo et al. [14] framed test smell detection as a supervised machine-learning task, at the cost of requiring labeled training data. The reliability of such detectors was questioned by Panichella et al. [13], who, by manually annotating hundreds of test suites, found smells that were ubiquitous in developer-written tests yet rarely matched real maintainability problems, with a widely used detector misclassifying over 70% of instances. This is a pertinent caveat for our study, whose ground truth is produced by a static linter rather than by manual inspection.

Closest to our scope, Ramos et al. [15] proposed dantes, a web tool that detects and refactors eleven test smell types in Java/JUnit code. dantes was evaluated only for perceived usefulness and ease of use, through a Technology Acceptance Model survey with 17 practitioners, and not for the correctness of the refactorings it produces. In contrast, we assess refactoring correctness directly and observe that detection and refactoring behave as decoupled capabilities, an asymmetry that a combined, hand-crafted pipeline leaves unmeasured.

7 Conclusion

This study investigated the extent to which three widely adopted large language models, Claude, GPT, and Gemini, are able to detect and refactor four performance-related test smells in real-world k6 load testing scripts, and how this capability is affected by the

presence of illustrative code examples in the prompt. The evaluation was conducted on a dataset of 228 k6 scripts, sampled from 58 GitHub repositories and annotated using the `k6-performance` linter [3]. Each script was submitted to each LLM under both prompt variants, totaling 1,368 automated queries, complemented by a manually reviewed sample of 120 refactoring suggestions. Three main findings emerge.

First, detection performance was moderate and broadly comparable across the three LLMs under the description-only prompt (P1), with F1 micro scores between 0.5273 and 0.5427 and accuracy between 0.6064 and 0.6283; Gemini achieved the most balanced detection profile overall, particularly excelling at S1 (costly init operations), with S4 (missing tags) performance on par with the other models, while all three models struggled markedly with S2 (missing thresholds) and S3 (missing checks). Second, contrary to our initial expectation, augmenting the prompt with illustrative code examples (P2) did not consistently improve detection: it degraded overall performance for Claude and GPT and produced a narrow, S1-concentrated gain for Gemini, suggesting that worked examples can anchor a model's attention on a single, easily pattern-matched smell at the expense of the others. Third, and most notably, the qualitative review revealed that detection difficulty and refactoring difficulty are largely decoupled: smells with the lowest detection F1 (S2, S3) were refactored almost flawlessly once identified, whereas S1, among the best-detected smells (with detection F1 up to 0.8571), accounted for the overwhelming majority of Incorrect and Breaking refactorings observed in the study. As discussed in Section 5, this asymmetry is consistent with a distinction between *declarative* smells, whose correction is a self-contained, well-documented configuration change, and *architectural* smells, whose correction requires reasoning about k6's execution lifecycle.

Future work should explore the detection and refactoring of performance test smells beyond the four categories investigated here, as a broader catalog may reveal LLM behavior patterns not captured in this study. Involving a second independent reviewer to assess refactoring quality would also strengthen the reliability of the qualitative analysis. Additionally, investigating alternative prompt strategies, such as chain-of-thought or few-shot prompting with positive and negative examples, could help mitigate the attentional anchoring effect observed with Prompt 2. Finally, extending this evaluation to other load testing frameworks and incorporating agentic methods [12] to handle larger and more diverse datasets are promising directions for future research.

Artifact Availability

The replication package is available at https://github.com/AdelsonOrnellas/Can-LLMs-Detect-and-Refactor-Test-Smells-in-Performance-Tests_ARTIFACT

Acknowledgements

Special thanks to Caio Barbosa, Ph.D., for his help. This work was financed in part by FAPEAL (Research Support Foundation of the State of Alagoas), Process E:60030.0000000591/2026. The work was also partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) https://www.ines.org.br, CNPq grant 408817/2024-0. The work was also partially supported by CNPq, grants 312195/2021-4, 404825/2023-0, and 443393/2023-0.

References

[1] 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 *Evaluation and Assessment in Software Engineering*. ACM, 170–180.

[2] Manoel Aranda, 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 *Evaluation and Assessment in Software Engineering*. ACM, 7–16.

[3] Samuel Barbosa. 2025. *Detecção de Test Smells em Testes de Carga Implementados no Framework K6*. Bachelor's Thesis. UFAL. Artifacts available at: https://github.com/samurollie/TCC.

[4] Dave Bergmann. 2025. *A List of Large Language Models (LLMs)*. Retrieved June 13, 2026 from https://www.ibm.com/think/topics/large-language-models-list

[5] Jonathan Cordeiro, Shayan Noei, and Ying Zou. 2026. An Empirical Study on the Code Refactoring Capability of Large Language Models. *ACM Transactions on Software Engineering and Methodology* (2026). doi:10.1145/3801158

[6] Grafana Labs. 2025. *k6 Documentation*. Retrieved June 20, 2026 from https://grafana.com/docs/k6/latest/

[7] Zhen Ming Jiang and Ahmed E. Hassan. 2015. A Survey on Load Testing of Large-Scale Software Systems. *IEEE Trans. on Soft. Eng.* 41, 11 (2015), 1091–1118.

[8] Gustavo Lopes, Davi Romão, Elvys Soares, Márcio Ribeiro, Guilherme Amaral, Rohit Gheyi, and Ivan Machado. 2024. A Road to Find Them All: Towards an Agnostic Strategy for Test Smell Detection. In *Brazilian Symposium on Software Quality*. ACM, 231–241.

[9] Keila Lucas, Rohit Gheyi, Márcio Ribeiro, Fabio Palomba, Luana Martins, and Elvys Soares. 2025. Investigating the Performance of Small Language Models in Detecting Test Smells in Manual Test Cases. In *Brazilian Symposium on Software Engineering*. SBC, 783–789.

[10] Keila Lucas, Rohit Gheyi, Márcio Ribeiro, Fabio Palomba, Luana Martins, and Elvys Soares. 2026. An Empirical Study of Gemini 3 for Detecting Natural Language Test Smells in Manual Test Cases. *Journal of Software Engineering Research and Development* (2026).

[11] Keila Lucas, Rohit Gheyi, Elvys Soares, Márcio Ribeiro, and Ivan Machado. 2024. Evaluating Large Language Models in Detecting Test Smells. In *Brazilian Symposium on Software Engineering*. SBC, 672–678.

[12] Rian Melo, Pedro Simões, Rohit Gheyi, Marcelo d'Amorim, Márcio Ribeiro, Gustavo Soares, Eduardo Almeida, and Elvys Soares. 2026. Agentic LMs: Hunting Down Test Smells. *IEEE Software* 43, 1 (2026), 32–40.

[13] 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.

[14] 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), 55.

[15] Daniel Bruno Ramos, Railana Santana, and Ivan Machado. 2025. DANTES: Automating Detection and Refactoring of Test Smells. In *Brazilian Symposium on Software Engineering (SBES'25) – Tools Track*. SBC, 1003–1009.

[16] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2024. *A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications*. arXiv:2402.07927

[17] Elvys Soares, Manoel Aranda, Naelson Oliveira, Márcio Ribeiro, Rohit Gheyi, Emerson Souza, Ivan Machado, André Santos, Baldoino Fonseca, and Rodrigo Bonifácio. 2023. Manual Tests Do Smell! Cataloging and Identifying Natural Language Test Smells. In *Emp. Soft. Engineering and Measurement*. IEEE, 1–11.

[18] Elvys Soares, Márcio Ribeiro, Guilherme Amaral, Rohit Gheyi, Leo Fernandes, Alessandro Garcia, Baldoino Fonseca, and André L. M. Santos. 2020. Refactoring Test Smells: A Perspective from Open-Source Developers. In *Brazilian Symposium on Systematic and Automated Software*. ACM, 50–59.

[19] Elvys Soares, Márcio Ribeiro, and André Santos. 2024. A Multimethod Study of Test Smells: Cataloging Removal and New Types. In *Brazilian Symposium on Software Quality*. SBC.

[20] Gabriela Soares, Vanessa Santos, Márcio Ribeiro, Luana Almeida Martins, Valeria Pontillo, Manoel Aranda III, Rohit Gheyi, Ivan Machado, and Fabio Palomba. 2025. On the Harmfulness of Test Smells in Manual System Testing: A Controlled Experiment. In *Empirical Software Engineering and Measurement*. IEEE, 185–195.

[21] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. 2001. Refactoring Test Code. In *Extreme Programming and Flexible Processes in Software Engineering*. 92–95.