Journal of Software Engineering Research and Development, 2026, 14:20, doi: 10.5753/jserd.2026.7576

An Empirical Study of Gemini 3 for Detecting Natural Language Test Smells in Manual Test Cases

Keila Lucas [**Universidade Federal de Campina Grande** | ***keila.santos@copin.ufcg.edu.br***]
Rohit Gheyi [**Universidade Federal de Campina Grande** | ***rohit@dsc.ufcg.edu.br***]
Márcio Ribeiro [**Universidade Federal de Alagoas** | ***marcio@ic.ufal.br***]
Fabio Palomba [**University of Salerno** | ***fpalomba@unisa.it***]
Luana Martins [**University of Salerno** | ***lalmeidamartins@unisa.it***]
Elvys Soares [**Instituto Federal de Alagoas** | ***elvys.soares@ifal.edu.br***]

Abstract

Manual testing, in which testers follow natural language instructions to validate system behavior, remains essential for uncovering issues that are difficult to capture with automation. However, manual test cases often contain *test smells*, quality issues such as ambiguity, redundancy, or missing checks that reduce reliability, maintainability, and reproducibility. Existing detection approaches largely depend on manually engineered rules and thus struggle to generalize and scale across heterogeneous test suites. In our previous work, we assessed the feasibility of using Small Language Models (SLMs) for test smell detection by evaluating Gemma-3-4B, Llama-3.2-3B, and Phi-4-14B on test steps from 143 real-world Ubuntu test cases, covering seven smell types. Phi-4-14B achieved the best performance. In this article, we investigate whether a contemporary Large Language Model (Gemini-3-Pro-Preview) available at the time of the study can identify test smells in natural language manual test cases using a prompt-based, whole-test-case analysis strategy. Unlike approaches that analyze individual test steps in isolation, our approach evaluates complete test cases, enabling the model to consider relationships and dependencies among test steps. We evaluate the approach on 100 Ubuntu test cases covering seven test smell types and compare its performance against previously evaluated SLMs, including Gemma-3-4B, Llama-3.2-3B, and Phi-4-14B. Our results show that Gemini-3-Pro-Preview outperforms the SLMs, while producing actionable explanations that can help practitioners revise manual test cases for greater clarity and consistency. We also find that test smells are pervasive in practice, with nearly one detected test smell per step on average, highlighting the need for scalable and automated quality support for manual testing artifacts.

Keywords: *Manual Testing, Test Smells, Large Language Models, Detection*

1 Introduction

Manual testing is a widely used software verification technique in which testers execute predefined steps to uncover defects in the system (Hauptmann, 2016; Soares et al., 2023a). In practice, these steps are typically documented in natural language and describe how to interact with the system and what to observe as evidence of correct behavior. Even in organizations with mature automated testing pipelines, manual testing remains essential for scenarios that are difficult to automate or validate reliably with scripts, such as exploratory workflows, complex user interactions, and environment-dependent behaviors (Hauptmann et al., 2013).

Despite its importance, this testing approach often suffers from quality and reliability issues. Manual test descriptions are frequently authored as operational documentation rather than as carefully engineered artifacts, and they may be created and updated outside formal software engineering practices (Hauptmann et al., 2013). As a consequence, manual tests often contain *test smells*; that is, quality problems that make tests harder to understand, maintain, and execute consistently. Prior work reports a broad range of such issues, including ambiguity, redundancy, unnecessary complexity, missing checks, and checks that are too weak or underspecified (Hauptmann et al., 2013; Juhnke et al., 2021; Junior et al., 2021, 2020; Soares et al., 2023a). For instance, the *Ambiguous Test* smell occurs when a step provides vague instructions, such as *"Select a keyboard layout and click Continue"*, without specifying which layout should be chosen or what constitutes the correct selection. In this case, different testers may legitimately interpret the same step in different ways, leading to inconsistent test execution and reduced confidence in the test results (Soares et al., 2025).

Improving the clarity and specificity of manual tests is challenging in practice. Writing high-quality natural language instructions requires domain knowledge, testing expertise, and careful attention to phrasing, yet test suites evolve continuously with the system. As a result, manual test repositories can accumulate quality issues over time, and systematically identifying them becomes a recurring maintenance task. Recent empirical studies have shown that smells in manual tests are not merely cosmetic, but can hinder execution and reduce efficiency. For example, Soares et al. (2025) report evidence that test smells can negatively affect the efficiency of manual test execution. Other studies highlight recurring problems such as ambiguous descriptions, excessive complexity, actions without checks, and inadequate checks, which can undermine the effectiveness of manual testing efforts (Soares et al., 2023a; Peixoto et al., 2025). These findings reinforce the relevance of practical techniques that help teams continuously assess and improve the quality of manual tests at scale.

However, existing automated support remains limited.

Many approaches for detecting smells in natural language rely on manually engineered rules, keyword lists, or heuristics tailored to specific writing conventions. While effective in narrow settings, such approaches typically require substantial implementation effort, may be brittle under vocabulary or style variation, and do not scale well across heterogeneous test suites or evolving projects. This limitation is particularly problematic because manual tests are often maintained by different contributors over long periods, which naturally leads to diverse writing styles and inconsistent levels of detail. A previous approach proposed a tool-based method using natural language processing to detect manual test smells (Soares et al., 2023a). However, it does not perform richer semantic analysis.

In parallel, the rapid evolution of foundation models has opened new opportunities for analyzing and improving natural language artifacts in software engineering, providing richer semantic analysis. In our previous study (Lucas et al., 2025), we investigated the feasibility of using Small Language Models (SLMs) to detect seven types of test smells in manual test descriptions (Aranda et al., 2024). We evaluated `Gemma-3-4B`, `Llama-3.2-3B`, and `Phi-4-14B` on a dataset of 261 test steps extracted from 143 manual test cases of the Ubuntu operating system, focusing on smells related to ambiguity, incomplete checks, unnecessary complexity, and redundancy. The results show that SLMs can detect test smells in practice, with `Phi-4-14B` achieving the best overall performance and the most stable behavior across temperature settings.

In this article, we investigate whether contemporary Large Language Models (LLMs), available at the time of the study, can provide scalable support for identifying test smells in natural language manual tests. Our key idea is to analyze complete test cases rather than isolated test-case steps. Whole-test-case analysis allows the model to leverage step-level context, capture dependencies between actions and verifications, and detect smells that only become evident when considering the surrounding steps. This setting is closer to how practitioners read and review manual tests, and it enables explanations that can be grounded in the broader flow of the procedure. We instantiate this idea using `Gemini-3-Pro-Preview` with an improved prompting strategy and evaluate it on 100 Ubuntu test cases, covering a substantially larger set of test-case steps and seven test smell types.

We evaluate `Gemini-3-Pro-Preview` under this whole-test-case setting. Our results show that `Gemini-3-Pro-Preview` consistently outperforms the SLM baselines and generates explanations that are actionable for revising manual tests. Beyond predictive performance, we provide evidence that test smells are pervasive in practice, with nearly one test smell per test-case step on average. This highlights that test smell detection is not a corner case, but rather a recurring maintenance concern in real test repositories.

These findings have practical implications for both researchers and practitioners. For practitioners, they suggest that LLM-based assistants can support systematic review of manual tests at scale, improving clarity and consistency while reducing manual inspection effort. For researchers, our results motivate further work on context-aware test quality analysis, robust prompting strategies, and hybrid approaches that combine model-based reasoning with lightweight checks and tooling integration. All data and artifacts are publicly available online (Lucas et al., 2026).

This article is organized as follows. Section 2 describes the natural language test smells considered in this study. Section 3 details our methodology. Section 4 presents and discusses the results. Section 5 situates our work with respect to related studies. Finally, Section 6 concludes the article and outlines implications and future work.

2 Natural Language Test Smells

Manual test cases are often written as natural language procedures that guide testers through actions and expected outcomes. Prior work shows that these artifacts frequently suffer from *natural language test smells*, that is, recurring quality issues that reduce clarity, consistency, and maintainability, and can ultimately compromise the reliability of manual testing outcomes (Hauptmann et al., 2013; Soares et al., 2023a, 2025; Peixoto et al., 2025; Aranda et al., 2024). In this article, we focus on seven smells that are both widely discussed in the literature and directly operationalized in our prompting strategy: Ambiguous Test, Conditional Test, Eager Action, Misplaced Action, Misplaced Precondition, Misplaced Verification, and Unverified Action (Hauptmann et al., 2013; Soares et al., 2023a; Aranda et al., 2024). These smells are particularly relevant because they can be identified from the text of test steps and their role in the test structure, and they directly impact test determinism and the ability of practitioners to follow and interpret tests consistently (Soares et al., 2025).

This section provides the background necessary to understand our study. We first introduce the structure of Ubuntu-style manual test cases in Section 2.1. We then present the seven test smells analyzed in this work: Ambiguous Test (Section 2.2), Conditional Test (Section 2.3), Eager Action (Section 2.4), Misplaced Action (Section 2.5), Misplaced Precondition (Section 2.6), Misplaced Verification (Section 2.7), and Unverified Action (Section 2.8). Finally, Section 2.9 discusses the practical relevance of these smells using evidence from Ubuntu manual-test maintenance. These sections provide the basis for both the manual annotation process and the prompting strategy evaluated in this work.

2.1 Ubuntu-Style Manual Tests

In this article, we focus on manual test cases from the Ubuntu ecosystem (Ubuntu, 2026). Ubuntu is one of the most widely used Linux distributions across desktop, cloud, and IoT settings.[1] We chose Ubuntu for three complementary reasons. First, Ubuntu provides a large, publicly accessible corpus of human-authored manual test cases, curated in a consistent, semi-structured format, which supports transparent and reproducible experimentation (e.g., the public `ubuntu-manual-tests` repository). Second, restricting the study to a single ecosystem reduces confounding variation (e.g., domain-specific terminology, documentation conventions, and test templates), helping us isolate language-level

[1] `https://canonical.com/blog/ubuntu-20-04-survey-results`

test smells rather than differences in writing styles across projects. Third, Ubuntu test cases are authored and maintained by a broad community and cover diverse end-user workflows, which mitigates single-author bias and strengthens ecological validity. We view this scope as a conservative first step; evaluating additional domains (e.g., mobile apps, web systems, or API-oriented documentation) is left as future work.

Ubuntu-style manual tests commonly separate what the tester *does* from what the tester *checks*. In Ubuntu's manual test specifications, the procedure is often encoded using an HTML-like list structure. A test case is represented as a `<dl>` block, where each `<dt>` element denotes an action step and each `<dd>` element denotes a verification step associated with the most recent action sequence. When a test case requires a prerequisite state or setup condition before execution, this information should be specified in the precondition section of the manual test case, before the executable procedural steps encoded with `<dt>` and `<dd>`. In our dataset of manual tests, however, preconditions are not explicitly marked with a dedicated HTML tag, unlike the `<dt>` and `<dd>` elements. Instead, the setup description or prerequisite conditions usually appear separately before the executable steps, either as free text or as an introductory paragraph preceding the first test action. Therefore, preconditions describe the state that must already hold before the tester begins execution, while the `<dt>` and `<dd>` entries represent, respectively, the tester's actions and the expected observations during execution.

This representation makes the intended role of each step explicit and supports a structured reading of the test: actions describe interactions with the system under test, while verifications describe the expected observable outcomes.

This separation is crucial for two reasons. First, several smells are defined by *role mismatches*, such as an action written as a verification or a verification written as an action. Second, the absence of an expected outcome after an action is itself a smell, because it leaves the tester without guidance about what constitutes correct behavior. In our study, we therefore interpret `<dt>` entries as *action* steps and `<dd>` entries as *verification* steps, using this structural information both to contextualize smell definitions and to enable the detection of smells that depend on the presence or absence of verifications.

Example. Listing 1[2] illustrates the Ubuntu-style encoding, in which preconditions are listed first, followed by action and verification elements encoded as `<dt>` and `<dd>`, respectively. A verification may state the expected outcome of multiple preceding actions.

2.2 Ambiguous Test

Definition. Ambiguous Test occurs when a step is vague or underspecified, leaving room for multiple interpretations and preventing an objective pass or fail decision (Hauptmann et al., 2013; Soares et al., 2023a; Aranda et al., 2024). Ambiguity often stems from indefinite determiners, unclear references, subjective qualifiers, or comparative claims without measurable criteria.

Why it is harmful. Ambiguity makes manual testing nondeterministic. Two testers can follow the same step and legitimately choose different inputs, paths, or interpretations of success, producing inconsistent outcomes and lowering confidence in the test suite (Hauptmann et al., 2013; Soares et al., 2025).

Examples. The following step is ambiguous because it does not specify which layout should be selected, which can lead testers to make different choices.

```
<dt>Select a keyboard layout and click
Continue</dt>
```

In the next example, the expected outcome is described using subjective language, and the test does not define concrete observable criteria for correctness.

```
<dt>Press 'Apply'. Verify that the new
resolution is set correctly and that the
screen is still visible.</dt>
```

2.3 Conditional Test

Definition. Conditional Test appears when a step embeds branching logic, such as conditions expressed with markers like if, when, unless, or otherwise (Hauptmann et al., 2013; Aranda et al., 2024). The presence of conditionals suggests multiple flows inside a single step or uncertainty about whether the step applies.

Why it is harmful. Conditionals introduce divergence and can reduce repeatability. Testers may not know whether they should execute alternative actions, skip steps or stop the test. This can lead to incomplete coverage or inconsistent execution across environments and testers (Hauptmann et al., 2013; Soares et al., 2023a).

Examples. The following step creates two possible states and does not clarify how the test should proceed in each case.

```
<dt>Press the 'Super Windows key', or click
the Ubuntu logo in the upper left hand unity
panel and search 'Sound Recorder'</dt>
```

In the following example, the step depends on a prior condition: the software must have been successfully installed before the tester can log out and verify the language selection options available on the login screen.

```
<dt>When they have been successfully
installed, log out from the session, and
select alternating languages at the login
screen, next to the right of the session
selector (Xubuntu Session)</dt>
```

2.4 Eager Action

Definition. Eager Action occurs when a single step groups multiple distinct actions that should be separated (Hauptmann et al., 2013; Aranda et al., 2024). In manual tests, this often appears as multiple imperative verbs connected by

[2]Manual test case #1442_Mythbuntu Frontend (image)

Listing 1. Ubuntu manual test case.

```
The testcase tests the installation and basic functionality of a Frontend.
You will need a MythTV backend in the network to do this testcase completely.

<dl>
    <dt>Go through the Ubiquity install as you would any other install. [...]</dt>
    [...]
    <dt>Once the machine boots up, it should boot into the frontend</dt>
    <dt>Select "Watch TV". (this will only work if you have a backend already in the network)</dt>
        <dd>The demo video should start playing</dd>
</dl>
```

"and", "then", commas, or multiple sentences within a single step.

Why it is harmful. Bundling multiple actions into one step reduces granularity and makes debugging difficult. If a later verification fails, the tester may not know which action caused the failure. It also increases cognitive load and encourages skipping or partially executing instructions (Soares et al., 2023a, 2025).

Examples. The following step groups several UI interactions into a single instruction, which makes it harder to isolate failures or confirm that each sub-action was completed correctly.

```
<dt>Open the dash and launch rhythmbox by
pressing the super key, and then entering
'rhythmbox'</dt>
```

In the next example, two sentences still form one step and bundle multiple operations that would be clearer as separate steps.

```
<dt>Select a plugin and configure it by doing
 click on "Configure"</dt>
```

2.5 Misplaced Action

Definition. Misplaced Action occurs when an action instruction is written in the verification field, that is, in a step whose role is to state expected outcomes (Soares et al., 2023a; Aranda et al., 2024). The key signal is imperative action language inside a verification context.

Why it is harmful. This smell blurs the separation of responsibilities. A verification step should describe what to observe, not what to do. Role confusion increases the likelihood of missing checks, repeating actions, or misreporting failures (Soares et al., 2025).

Examples. The following text is an action instruction appearing in a verification step. In this case, the test mixes procedure and oracle information in the wrong field.

```
<dd>Open some windows (e.g. terminal)</dd>
```

2.6 Misplaced Precondition

Definition. Misplaced Precondition occurs when a prerequisite or required state is written as an action or verification step, instead of being stated explicitly as a precondition (Soares et al., 2023a; Aranda et al., 2024). Common patterns include state descriptions such as the user is logged in, make sure, assuming, given that, or ensure that followed by a state rather than an interaction.

Why it is harmful. Preconditions define when a test is applicable. When they are embedded as ordinary steps, testers may interpret a missing prerequisite as a product failure, which can lead to false bug reports and wasted effort during triage (Hauptmann et al., 2013; Soares et al., 2025).

Examples. The following sentence describes a precondition state appearing as an action step. In this case, it should be extracted into an explicit precondition.

```
<dt>Please ensure the Nautilus window is
running and the Nautilus window is focused
</dt>
```

The next example is a state assertion rather than an action or a verification derived from an action, which indicates that it is a precondition expressed in the wrong location.

```
<dt>Ensure that Network Manager is running
and that no networks are currently connected
</dt>
```

2.7 Misplaced Verification

Definition. Misplaced Verification occurs when a verification or expected outcome is written in the action field (Soares et al., 2023a; Aranda et al., 2024). Typical cues include verbs such as verify, check, confirm, ensure, or outcome statements with should, must, is displayed, appears, or no error.

Why it is harmful. Mixing checks into action steps can lead testers to treat verification as optional or to perform checks at the wrong time. It also complicates automated processing because the same field contains both procedure and oracle information (Hauptmann et al., 2013; Soares et al., 2025).

Examples. The following step is phrased as a check and appears in the action field. In this case, it is a verification written in the wrong location.

```
<dt>Run 'apt-get update' and verify that an
ec2 mirror is used</dt>
```

The next example states an expected outcome rather than an interaction and is written in the action field. Therefore, it should be expressed as a verification step.

Table 1. Catalogued test smells considered in this article.

Test Smell	Brief definition
Ambiguous Test	Test step is vague or underspecified, leaving room for interpretation.
Conditional Test	Step includes conditional logic expressed in natural language.
Eager Action	Single step groups multiple distinct actions that should be separated.
Misplaced Action	Action instruction written as a verification step.
Misplaced Precondition	Prerequisite or required state written as an action or verification step.
Misplaced Verification	Verification or expected outcome written as an action step.
Unverified Action	Action step without a corresponding verification describing expected outcome.

```
<dt>Once the machine boots up, it should boot
 into the frontend</dt>
```

2.8 Unverified Action

Definition. Unverified Action occurs when an action step has no corresponding verification that states what the tester should observe after performing that action (Hauptmann et al., 2013; Aranda et al., 2024). In structured manual tests, this is often identified by an action that is not followed by any verification step before the next action begins.
Why it is harmful. Without an explicit oracle, the tester cannot determine whether the action succeeded or whether the system behaved correctly. This invites implicit assumptions and increases variability across testers, which reduces test reliability and weakens the evidential value of manual execution (Soares et al., 2025).
Examples. The following instruction describes an interaction but provides no indication of what should be observed next. Without a subsequent verification step, the action is effectively unverified.

```
<dt>Click the Search Icon in the top</dt>
<dt>Input 'test'</dt>
    <dd>the 'test' folder was shown?</dd>
```

In the next example, the test instructs the tester to open the application. Without an immediate verification describing the expected screen or status, the notion of success remains underspecified.

```
<dt>Open the file manager</dt>
```

Table 1 summarizes the seven smells targeted by our prompt and used in our evaluation. The definitions are concise by design, and the preceding subsections provide rationale and concrete examples to support consistent interpretation.

2.9 Practical Relevance

Evidence from the Ubuntu manual-tests repository suggests that the issues captured by our catalogue of test smells arise in practice and are actively addressed during routine test maintenance. For instance, in a bug-fix commit (Fix bug #2012346), maintainers refined a manual test case by clarifying the instructions, splitting a previously coarse step into multiple, more focused test steps, and adding an explicit expected outcome (i.e., a verification that UI elements appear in the selected language).[3] This revision improves precision and reduces the risk of underspecified or unverified actions. At the same time, the patch also illustrates how smells can be inadvertently introduced during editing: the following newly added step expresses a check but is recorded as an *action* step (i.e., it appears in a `<dt>` field), which corresponds to our *Misplaced Verification* category.

```
<dt>[...] and verify that an ec2 mirror is
used</dt>
```

Overall, such changes likely reflect manual review and iterative refinement by practitioners, yet they also highlight the lack of lightweight support for reasoning about the quality of natural language test cases. Our goal is to automate the detection of these issues, enabling test authors and reviewers to identify ambiguous, conditional, misplaced, eager, or unverified steps earlier and with lower effort.

3 Methodology

This section describes the methodology adopted to evaluate the ability of foundation models to detect natural language test smells in Ubuntu-style manual test cases. We define our study goals and research question (RQ) following the Goal Question Metric (GQM) paradigm in Section 3.1. We then present the dataset and annotation procedure in Section 3.2, describe the prompting strategy used to operationalize the test smell definitions in Section 3.3, and detail the evaluated models and evaluation procedure in Sections 3.4 and 3.5, respectively.

3.1 Goal Question Metric

We structure our study using the Goal Question Metric (GQM) paradigm (Basili et al., 1994). Our goal is to evaluate whether a contemporary LLM can accurately identify test smells in manual test descriptions while providing explanations that are useful to practitioners.
Goal. Analyze Gemini-3-Pro-Preview for the purpose of evaluating its effectiveness in detecting natural language test smells in Ubuntu-style manual test cases with respect to detection performance and explanation usefulness from the viewpoint of software testers and practitioners in the context of real-world manual testing artifacts.
Based on this goal, we derive the following RQ:

RQ$_1$ To what extent can Gemini-3-Pro-Preview detect natural language test smells in Ubuntu-style manual test cases?

[3] `https://git.launchpad.net/ubuntu-manual-tests/commit/?id=def6df4d810dedeea53fc3a63796aa2c717b33fb`

Metrics. To answer RQ_1, we report: (i) aggregated statistics on smell prevalence, including the average number of detected smells per test-case step, to characterize their frequency in practice; and (ii) detection performance using standard classification metrics, namely precision, recall, and F_1 score, computed per smell type and overall. Our ground truth labels were obtained through manual annotation: from the 2,618 test steps in our dataset, one author annotated a subset of 103 test steps from 15 test cases following the definitions detailed in Section 2. Additionally, we analyze explanation usefulness by verifying whether model-generated explanations identify concrete textual triggers aligned with the smell definitions.

3.2 Dataset

Our dataset is drawn from the Ubuntu Manual Tests repository (Ubuntu, 2026), which contains community-maintained manual test cases used by the Ubuntu QA (Quality Assurance) team to support release and regression activities, including ISO image testing and broader system validation. Ubuntu is a widely used Linux distribution with a large global user base, and its QA infrastructure relies on these manual test specifications to coordinate execution and record results through instances of the QATracker platform (Hauptmann et al., 2013; Soares et al., 2023a). In this study, we analyze 100 manual test cases from this repository.

3.3 Prompting Strategy

We employ Meta Prompting (Hou et al., 2022; DAIR.AI, 2024) to guide the model in generating and refining our task prompt. We used ChatGPT 5.2 Thinking in January 2026 to improve the prompt used before (Lucas et al., 2025). Then, we used Gemini-3-Pro-Preview to improve the prompt generated by ChatGPT 5.2 Thinking. The resulting prompt combines a domain-specific persona (QA engineer and test smell detector), explicit instructions and constraints, and a small set of few-shot examples to calibrate the desired behavior and output format. This process results in the final prompt, which we present next.

You are an expert QA Engineer and Natural Language Test Smell Detector.

TASK
Analyze ONE complete manual test case (Ubuntu manual test style) and detect Natural Language Test Smells for EACH step, based ONLY on the step text and the step location derived from the markup:
- `<dt> ... </dt>` ⇒ `where = "action"`
- `<dd> ... </dd>` ⇒ `where = "verification"`

IMPORTANT INPUT SHAPE
You will receive the full test case as a single text block that may contain:
- A title or intro line (plain text)
- An HTML like `<dl>` list with multiple `<dt>` and `<dd>` elements
- Other HTML tags (e.g., `<strong>`, `<a>`, etc.) that are NOT steps

You must:
1) Extract ONLY the text inside each `<dt>` and `<dd>` (strip tags inside them, keep the visible text).
2) Treat each `<dt>` or `<dd>` as ONE step item, even if it contains multiple sentences.
3) Ignore any content outside `<dl> ... </dl>` for smell detection (including instructions like `If all actions produce...`).

OUTPUT (STRICT)
Return ONLY a valid JSON object (no markdown, no extra text) with EXACTLY these fields:

```
{
"steps": [
{
"id": 1,
"where": "action",
"text": "Step text as plain text",
"smells": ["Smell Name 1", "Smell Name 2"],
"explanation": "Concise reasoning citing the
exact words or patterns that triggered the
smell(s)."
}
]
}
```

RULES FOR THE JSON OUTPUT
- `"id"` must start at 1 and increment by 1 following the order in the `<dl>`.
- If no smells apply to a step:
`"smells": []`
`"explanation": "No clear smell indicators in the step given its field."`
- Do NOT add any other JSON fields (no title, no summary, no counts).

GENERAL RULES (STRICT)
- Be conservative: label a smell only when there is clear evidence in the step text.
- You may assign multiple smells to the same step.
- Always use the smell names EXACTLY as listed below.
- Explanations must be short (one or two reasons) and MUST cite exact triggers from the step text
(e.g., `contains "if"`, `contains "Verify"`, `contains "any"`, `multiple imperative verbs: "open", "click"`).
- If `where="action"` and the step mixes actions and verification language, label BOTH `"Eager Action"` and `"Misplaced Verification"`.
- Do NOT infer context outside the step text.
- Do NOT use surrounding steps to decide a smell for the current step,
EXCEPT for `"Unverified Action"`, which requires checking whether a `<dt>` action step
has at least one corresponding `<dd>` verification step associated to it by structure.

SPECIAL CASE FOR `"Unverified Action"`
For `"Unverified Action"` only, you may use the `<dl>` order to check whether a `<dt>` action step
is followed by a corresponding `<dd>` verification step (before the next `<dt>`).
For all other smells, do not use surrounding steps.

SMELLS AND RULES (STRICT)

1) `"Ambiguous Test"`
Definition: Vague, subjective, or underspecified step; pass or fail is not objectively verifiable.
Mark if ANY trigger applies:
A) Verb + indefinite determiner or vague quantity targeting an open-ended object:
- Patterns like: `"Open any ..."`, `"Select a ..."`, `"Choose some ..."`, `"Pick several ..."`
- Indefinite determiners or quantities: `"any"`, `"a"`, `"an"`, `"some"`, `"a few"`, `"several"`, `"various"`, `"a lot"`
- IMPORTANT: Only mark this when the object is not uniquely

identified
(no clear label or identifier like quoted UI text, exact value, ID, or precise menu path).
B) Indefinite pronouns or unclear referent:
- "something", "someone", "somewhere", "anything", "anyone", "anywhere", "everything"
- "it/this/that" ONLY if the referent is unclear inside the same step text.
C) Subjective adverbs or adjectives or phrases:
- "as expected", "properly", "correctly", "user-friendly", "randomly", "normally", "adequately", "quickly/fast"
D) Comparative or superlative language without objective metric:
- "better", "worse", "faster", "more stable", "best", "most", "least"

2) "Conditional Test"
Definition: Introduces branching logic or depends on a condition, implying multiple flows.
Mark if the step contains conditional markers:
"if", "when", "unless", "in case", "depending on", "whether", "as long as", "otherwise", "only if"

3) "Eager Action"
Definition: A single step groups multiple distinct actions that should be separate.
Mark if there are two or more distinct operations, signaled by:
- multiple imperative verbs, and or connectors: "and", "then", commas, "after", "followed by"
- multiple sentences within the same step that contain multiple operations
Examples of action verbs: Open, Click, Select, Type, Navigate, Install, Run, Adjust, Press, Tap, Choose, Enter

4) "Misplaced Action"
Definition: Action instructions written in the verification field.
Mark ONLY IF:
- where == "verification"
- AND the step contains imperative action verbs (e.g., "Click", "Open", "Type", "Select", "Adjust", "Install", "Navigate")

5) "Misplaced Precondition"
Definition: Prerequisite or setup or state written in action or verification instead of precondition.
Mark ONLY IF:
- where in ["action","verification"]
- AND the step is primarily a prerequisite or state (not an action), e.g.:
- SUT state pattern: "X is/are/was/were/has/have ..." + adjective or past participle (e.g., "The user is logged in.")
- explicit cues: "Precondition:", "Prerequisite:", "Make sure", "Assuming", "Given that", "Ensure" (state)

6) "Misplaced Verification"
Definition: Verification or checking written in the action field.
Mark ONLY IF:
- where == "action"
- AND the step includes verification verbs or outcome assertions:
- verification verbs: "Verify", "Check", "Ensure", "Confirm", "Validate", "Observe"
- outcome language: "should", "must", "is displayed", "appears", "changes", "is updated", "is saved", "no error", "successfully"

7) "Unverified Action"
Definition: Action steps that miss corresponding verification steps.
Problem: Absent verification steps negatively affect test execution and correctness since there is no instruction on how the system should behave, leaving room for the testers' interpretation.
Identification (STRICT): Mark ONLY IF:
- where == "action"
- AND the step is an action instruction (contains imperative action verb like "Open", "Click", "Select", "Type", "Navigate", "Install", "Run", "Press", etc.)
- AND the step does NOT contain verification or outcome language by itself
(e.g., "Verify", "Check", "Ensure", "Confirm", "should", "is displayed", "appears", "no error", "successfully")
- AND there is NO corresponding verification step structurally linked to it:
- i.e., there is no <dd> step immediately following this <dt> before the next <dt>, in the extracted <dl> order.
Notes to reduce false positives:
- If the action step itself contains verification or outcome language, do NOT mark "Unverified Action".
Instead, if where == "action", mark "Misplaced Verification" and possibly "Eager Action" per the rules above.
- Do not search for a verification elsewhere in the test case. Only consider the immediately following <dd> as the correspondence rule.

FEW-SHOT EXAMPLES (FULL TEST CASE STYLE)

Input test case:
```
<dl>
<dt>Open the browser and navigate to
Google.</dt>
<dt>Select the Settings menu and click
Advanced.</dt>
<dd>The Advanced settings are displayed.</dd>
</dl>
```

Output:
```
{
"steps": [
{ "id": 1, "where": "action", "text": "Open the
browser and navigate to Google.", "smells":
["Eager Action","Unverified Action"],
"explanation": "Multiple imperative verbs:
'Open' and 'navigate' connected by 'and'. No
immediate <dd> verification follows this <dt>."
},
{ "id": 2, "where": "action", "text": "Select
the Settings menu and click Advanced.",
"smells": ["Eager Action"], "explanation":
"Multiple imperative verbs: 'Select' and
'click' connected by 'and'. An immediate <dd>
verification follows this <dt>." },
{ "id": 3, "where": "verification", "text": "The
Advanced settings are displayed.", "smells": [],
"explanation": "No clear smell indicators in the
step given its field." }
]
}
```

NOW ANALYZE THIS INPUT TEST CASE
```
<test_case>
{test}
</test_case>
```

For clarity, the phrase "not an action" is intended to exclude not only procedural actions but also verification statements and outcome assertions. Verification-oriented text is captured separately by the Misplaced Verification smell, whereas Misplaced Precondition is reserved for prerequisite states and setup conditions that should appear in the precondition section of the test case.

In the prompt, `test` represents the Ubuntu test case. For

example, `test` corresponds to the test case in Listing 1. To guide Gemini-3-Pro-Preview in detecting test smells, we designed a structured prompt that (i) embeds the definitions of the seven smells considered in this study, grounded in the definitions used in prior work (Hauptmann et al., 2013; Soares et al., 2023a; Aranda et al., 2024), and (ii) explicitly encodes the Ubuntu manual test representation. In particular, the prompt explains that Ubuntu test cases are provided as an HTML-like `<dl>` block, where each `<dt>` element corresponds to an *action* step (what the tester does) and each `<dd>` element corresponds to a *verification* step (what the tester checks). The model is instructed to extract only the text inside `<dt>` and `<dd>`, treat each extracted element as a single step item, and ignore any content outside the `<dl>` block. The prompt enforces a strict JSON schema with one entry per step, including the step role, the predicted smell labels, and a short explanation that cites concrete textual triggers (e.g., the presence of conditional markers, vague quantifiers, or multiple imperative verbs). Finally, the prompt includes a special rule for *Unverified Action*, allowing the model to use only the local `<dl>` order to determine whether an action step is followed by a corresponding verification step, while requiring all other smells to be decided solely from the current step text.

3.4 Model

We selected Gemini-3-Pro-Preview (Google) because it is widely used in practice and is consistently ranked among the most competitive models on LLM Arena (Chiang et al., 2024). We accessed the model through the official API and used the default inference settings provided by the API client. All experiments were conducted between January and February 2026.

3.5 Evaluation Procedure

For each test case, we run Gemini-3-Pro-Preview with our prompting strategy and obtain, for every test step, (i) a predicted set of smell labels and (ii) a short, evidence-grounded explanation. We evaluate a subset of these predictions by comparing the model-assigned labels against a manually established reference. Concretely, one author labeled the selected steps according to the smell definitions in Section 2; we treat these annotations as ground truth to compute precision, recall, and F_1 score. We consider a prediction correct if the model assigns *exactly* the same set of smell labels as the manual annotation for the corresponding step. In addition to label agreement, we qualitatively inspect explanations to verify whether they cite explicit textual cues (e.g., keywords, patterns, or phrasing) that are consistent with the corresponding smell definitions.

Given the cost of manual labeling, we manually inspected $n = 103$ test steps drawn from 15 test cases, out of a population of $N = 2,618$ steps. Under a finite-population assumption and worst-case proportion ($p = 0.5$), this sample size yields an approximate 95% margin of error of about ± 10 percentage points when estimating step-level proportions. To improve coverage under a limited labeling budget, we adopted a stratified sampling strategy that includes both smelly and non-smelly steps and prioritizes the most frequent smell categories. We therefore treat this manual assessment as an initial, cost-effective validation, and we discuss larger-scale annotation as future work.

4 Results

This section presents the results of our empirical evaluation. Section 4.1 characterizes the prevalence of test smells across the analyzed Ubuntu test cases and reports the smells identified by Gemini-3-Pro-Preview. Section 4.2 compares the model predictions against a manually annotated baseline to assess detection performance. We then compare Gemini-3-Pro-Preview with ChatGPT 5.2 Thinking in Section 4.3, discuss the relationship between our findings and previous results obtained with SLMs in Section 4.4, and compare our approach with the rule-based Manual Test Sensei tool in Section 4.5. Finally, Section 4.6 discusses the main threats to validity of our study.

4.1 RQ_1. Gemini-3-Pro-Preview

We report the results of manual test smell detection using Gemini-3-Pro-Preview as the underlying model. The analyzed corpus contains 100 manual test cases with 2,618 test steps in total. Across all steps, Gemini-3-Pro-Preview flagged 2,293 smell instances (i.e., a step may contain multiple smells). Out of the 2,618 steps, 1,533 steps (58.56%) contain at least one smell, while 1,085 steps (41.44%) contain no smells. This indicates that smell occurrences are common at the step level: more than half of the steps are affected by at least one natural language issue.

We quantify density in two complementary ways. First, the overall smell density is 0.876 smells per step. Second, considering only steps that exhibit at least one smell, Gemini-3-Pro-Preview reports an average of 1.496 smells per smelly step, indicating that multi-smell steps are common. Smell counts also vary substantially across test cases. The minimum is one smell, observed in `packages/1423_LibreOffice/libreoffice/lbc-004`.[4] This test case contains only two steps, and exactly one is flagged with Eager Action.

```
<dt>Open the dash and launch LibreOffice
Writer by pressing the super key, and then
entering 'writer'</dt>
```

Gemini-3-Pro-Preview produces the following output:

> ***Smells:*** *Eager Action*
> ***Explanation:*** *Multiple imperative verbs and actions: 'Open', 'launch', 'pressing', 'entering' connected by 'and', 'and then'.*

As another example, consider the test case `image/1461_MAAS (Installer)/maas/maas-001`.[5] This test case includes 13 test steps, and each step is

[4] Manual test case #1423 (packages)
[5] Manual test case #1461 (image)

annotated with up to three test smells. For instance, consider the second action step:

```
<dt>Please ensure you have full control over
the network</dt>
```

Gemini-3-Pro-Preview returns the following prediction:

> ***Smells:*** *Ambiguous Test; Misplaced Precondition; Misplaced Verification*
> ***Explanation:*** *Contains subjective phrase 'full control'. Step describes a prerequisite state ('Please ensure you have...'). Contains verification verb 'ensure' in the action field.*

In this case, the test case does not provide an explicit precondition field. As a result, Gemini-3-Pro-Preview cannot disambiguate whether the sentence should be modeled as (i) a precondition that was placed in an action step, or (ii) a verification statement that should appear as a `<dd>` but was mistakenly written as a `<dt>`. Consequently, Gemini-3-Pro-Preview reports both Misplaced Precondition and Misplaced Verification. It also flags Ambiguous Test, since the phrase "full control over the network" is underspecified and does not define an objective, verifiable condition.

At the step level, the maximum number of smells observed in a single test step is 4. At the test-case level, the maximum is 101 test smells, observed in `1460_iSCSI Authenticated Root Installation`.[6] This test case includes 51 test steps, and each step is annotated with one to three test smells. For instance, consider the following action step:

```
<dt>On Keyboard Layout select No and press
Enter</dt>
```

Gemini-3-Pro-Preview produces the following output:

> ***Smells:*** *Eager Action; Unverified Action*
> ***Explanation:*** *Multiple imperative verbs 'select' and 'press' connected by 'and'. No immediate <dd> verification follows this <dt>.*

In Figure 1, the number of test steps per test case varies moderately (median = 23), with a few longer outliers reaching 84–85 steps; similarly, total test smell instances per test case have median = 19.5 and show a heavy tail with outliers up to 101.

Complementarily, Figure 2 shows the distribution of absolute counts for each smell type across test cases. The distributions shown in Figure 2 indicate that Unverified Action, Ambiguous Test, Eager Action, Misplaced Verification, and Conditional Test are recurrent across the analyzed test cases, whereas Misplaced Action and Misplaced Precondition tend to appear less frequently. The presence of several high-value outliers further indicates that some test cases accumulate substantially more smell instances than others. Together, Figures 1 and 2 suggest that natural language test smells are widespread throughout the corpus, although their distribution is uneven across both test cases and smell categories.

[6] Manual test case #1460 (image)

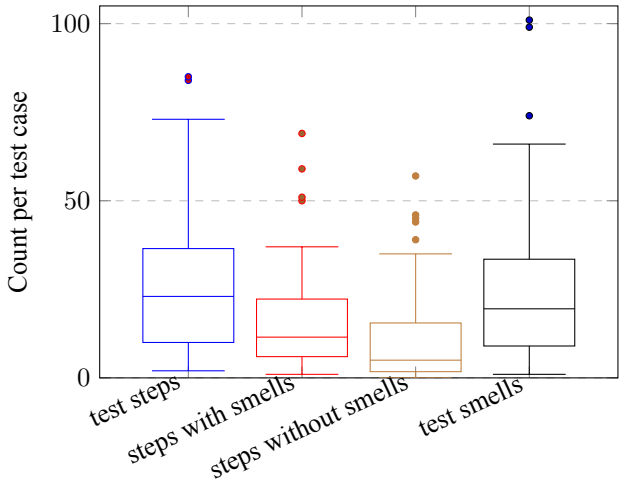

Figure 1. Aggregated absolute metrics per test case.

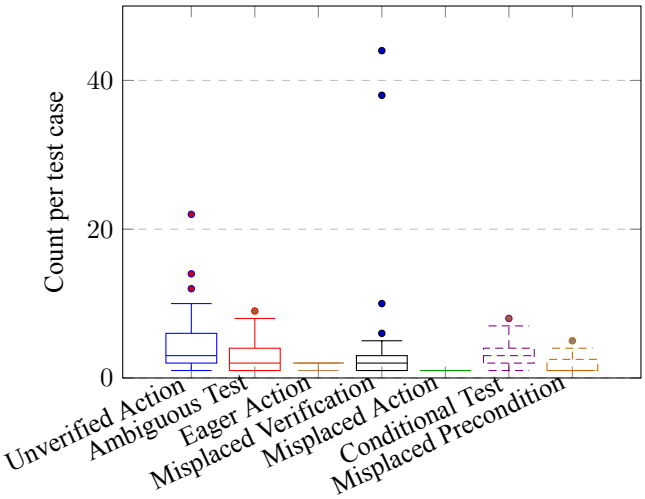

Figure 2. Absolute test smell counts per test case.

This spread suggests that some test cases are comparatively clean, whereas others concentrate a large number of issues and would likely require substantial rewriting to improve readability and verifiability. At the corpus level, the three most common smells are Unverified Action (580), Eager Action (563), and Ambiguous Test (465). See more details in Table 2. Together, these categories account for the majority of all detected smell instances, indicating that verification clarity, step granularity, and ambiguity are recurrent problems in the analyzed test steps.

Beyond counting how many times each smell occurs at the step level, we also measure *prevalence* at the test-case level: for each smell, we compute the percentage of test cases (out of 100) that contain at least one instance of that smell. Table 2 shows that test smells are widespread across the analyzed manual test cases. The most prevalent smells are Eager Action (94%) and Ambiguous Test (92%), indicating that most test cases include steps that either bundle multiple actions or rely on vague and subjective wording. Misplaced Verification is also frequent (76%), suggesting that verification statements are often written in the wrong field (e.g., expressed as actions). Mid-level prevalence smells include Conditional Test (65%) and Unverified Action (64%), highlighting that conditional branching and missing/insufficient verification occur in roughly two thirds of the test cases. In

Table 2. Prevalence of each test smell across 100 test cases (percentage of test cases with at least one instance) and total occurrences in all steps.

Test smell	Count	Prevalence (%)
Eager Action	563	94%
Ambiguous Test	465	92%
Misplaced Verification	303	76%
Conditional Test	200	65%
Unverified Action	580	64%
Misplaced Action	139	26%
Misplaced Precondition	43	23%

contrast, Misplaced Action (26%) and Misplaced Precondition (23%) are less common, implying that field misplacement affects a smaller subset of the corpus.

4.2 Manual Analysis

It is important to interpret our results against a baseline. However, since the dataset contains 2,618 test steps, manually reviewing all steps would be prohibitively time-consuming. Therefore, the first author manually analyzed 103 steps sampled from 15 test cases and used these annotations as a baseline. Using this baseline, our Gemini-3-Pro-Preview-based approach achieves 0.92 precision and 0.91 recall, with an F_1 score of 0.91.

Overall, Gemini-3-Pro-Preview correctly detected most smell categories. In some cases, the model identifies local cues (e.g., multiple verbs or embedded assertions) but misses the broader pattern of missing verification. For example, consider the test case `1482_Wireless Networking/hwn-001`.[7] This test case includes 11 steps. Consider the following action step:

```
<dt>Run <code>route -n</code>. Note the IP
address in the Gateway column. (not "0.0.0.0")
</dt>
```

The annotation produced by Gemini-3-Pro-Preview is as follows:

> ***Smells:*** *Eager Action; Misplaced Verification*
> ***Explanation:*** *Multiple imperative verbs: 'Run' and 'Note'. Contains outcome assertion '(not "0.0.0.0")'*

According to our baseline, these two smells are correctly identified. However, the step also exhibits Unverified Action: the instruction asks the tester to execute a command and inspect its output, yet the surrounding steps do not provide an explicit, dedicated verification (i.e., a clear `<dd>` check) that confirms the intended system behavior. In this case, Gemini-3-Pro-Preview does not flag Unverified Action, likely because the expected value is embedded inline rather than expressed as a standalone verification step, and because the subsequent step is also an action, which further obscures the missing verification structure.

A non-negligible portion of the disagreements with our baseline involved Ambiguous Test. For example, consider again the test case `1482_Wireless Networking/hwn-001` and the following action step:

```
<dt>After approximately 30 seconds, open the
network manager</dt>
```

Gemini-3-Pro-Preview does not flag any smell for this step. In contrast, our domain specialist labeled it as Ambiguous Test, because the waiting time ("approximately 30 seconds") is not precisely defined and may introduce variability in outcomes across executions or environments. This example also illustrates that Ambiguous Test can be inherently subjective: depending on the context, "approximately" may be acceptable operational guidance, yet it can also be interpreted as underspecification when timing differences can affect the observed behavior. Consequently, in borderline cases like this, it is not always straightforward to determine whether a step should be considered ambiguous.

4.3 ChatGPT 5.2 Thinking

Several other LLMs are available for this task. Among them, ChatGPT 5.2 Thinking has shown strong performance in community-driven evaluations such as LLM Arena (Chiang et al., 2024). In February 2026, we manually executed the same prompt described in Section 3.3 using ChatGPT 5.2 Thinking in thinking mode, with one temporary conversation per query to avoid cross-query contamination. We evaluated 103 test steps sampled from 15 test cases, for which the first author had previously established a manual baseline. Using this baseline, our ChatGPT 5.2 Thinking-based approach achieves 0.84 precision and 0.80 recall, with an F_1 score of 0.82. Overall, its performance is inferior to Gemini-3-Pro-Preview.

Overall, ChatGPT 5.2 Thinking correctly detected most smell categories, but it missed some cases.

We also revisited the two representative disagreement cases discussed in Section 4.2, where Gemini-3-Pro-Preview failed to reproduce the manual baseline. Interestingly, ChatGPT 5.2 Thinking correctly identified the smells missed by Gemini-3-Pro-Preview in both cases.

In test case `image/1303_Live Session (Test-case Live Session Start)`[8], ChatGPT 5.2 Thinking matched the baseline by flagging the step as ambiguous, whereas Gemini-3-Pro-Preview reported no smell. The analyzed step is shown next:

```
<dt>Boot up the image</dt>
```

ChatGPT 5.2 Thinking identifies the expression "the image" as under-specified:

> ***Smells:*** *Ambiguous Test*
> ***Explanation:*** *Target is under-specified: 'the image' is not uniquely identified (no name/ID/version/path).*

By contrast, Gemini-3-Pro-Preview does not flag the step:

[7] Manual test case #1482 (hardware)

[8] Manual test case #1303 (image)

Smells: *None*
Explanation: *No clear smell indicators in the step given its field.*

A similar situation occurred in `hardware/1482_Wireless Networking`[9], where ChatGPT 5.2 Thinking correctly identified ambiguity in a step that Gemini-3-Pro-Preview considered smell-free. The analyzed step is shown next:

```
<dt>After approximately 30 seconds, open the
network manager</dt>
```

ChatGPT 5.2 Thinking identifies the phrase "approximately 30 seconds" as vague timing, which may hinder consistent test execution:

Smells: *Ambiguous Test*
Explanation: *Uses vague timing "approximately 30 seconds", which is under-specified for consistent execution.*

By contrast, Gemini-3-Pro-Preview does not flag the step:

Smells: *None*
Explanation: *No clear smell indicators in the step given its field.*

Together, these examples suggest that, although Gemini-3-Pro-Preview achieves better aggregate performance, ChatGPT 5.2 Thinking may be more sensitive to subtle ambiguity and more closely aligned with human judgment in borderline cases.

4.4 SLMs

In our prior work (Lucas et al., 2025), we evaluated Gemma-3-4B (Team et al., 2025), Llama-3.2-3B (Touvron et al., 2023), and Phi-4-14B (Abdin et al., 2024) on 261 individual test steps extracted from 143 real-world Ubuntu manual test cases, targeting seven natural language test smells. Among the evaluated SLMs, Phi-4-14B achieved the best performance, reaching a $pass@2$ of 97% for detecting steps containing test smells, while Gemma-3-4B and Llama-3.2-3B achieved around 91%. Beyond accuracy, the results indicated that SLMs can provide a low-cost, concept-driven alternative to rule-heavy approaches, enabling smell identification without extensive syntactic analysis and offering practical advantages such as local deployment and improved data privacy.

In the present work, we extend this investigation in three key ways. First, we move from step-level inputs to analyzing complete test cases, which better reflects how manual tests are authored and reviewed, but also increases input length and contextual dependencies, potentially challenging smaller models. Second, we strengthen the prompting strategy by embedding the full set of smell definitions and explicit identification rules in the prompt, and by enforcing strict evidence-grounded explanations. Compared to our earlier SLM setting, this reduces the likelihood of guessing and encourages the model to align predictions with concrete textual cues and with the action versus verification roles implied by the Ubuntu `<dl>`/`<dt>`/`<dd>` structure. Third, we scale the analysis to 100 complete Ubuntu test cases, allowing us to better characterize smell prevalence in practice. While this LLM-based setting introduces higher inference cost and requires sending data to an external API, our results show that Gemini-3-Pro-Preview achieves excellent performance even with $pass@1$, and produces concise explanations that can support practitioners in improving the clarity and consistency of manual test cases.

Analyzing each test step in isolation, as in our prior work (Lucas et al., 2025), can lead to false positives because some smells depend on information that is only available at the test-case level. In such situations, considering the surrounding steps provides essential context. For example, consider the test case `hardware/1467_External Microphone`.[10] and the following action step:

```
<dt>Connect the microphone to the computer.
(This can include a microphone on a headset)
</dt>
```

The annotation produced by Gemini-3-Pro-Preview is as follows:

Smells: *Unverified Action*
Explanation: *Action step 'Connect' is not followed by a verification step and contains no verification language.*

Under a strictly step-level analysis, our previous approach would likely flag Ambiguous Test, since the referenced commands are not listed within the step itself. However, when inspecting the full test case, the missing details are provided in subsequent steps: the test case enumerates seven concrete commands to execute. This example highlights a key trade-off. Step-level processing can be useful for models with limited context windows (Brown et al., 2020), but it may also overlook cross-step dependencies and thereby produce incorrect smell assignments when critical information is distributed across the test case.

Overall, the comparison highlights a trade-off between SLMs and LLMs. SLMs offer lower inference cost, easier local deployment, and stronger privacy and control guarantees, but they may be more sensitive to prompt length and long-range context when the input grows from isolated steps to complete test cases. In contrast, LLMs such as Gemini-3-Pro-Preview tend to handle longer inputs and richer contextual dependencies more reliably and can generate more consistent, evidence-grounded explanations, but at the expense of higher runtime cost and reliance on external APIs, which may raise data governance and reproducibility concerns.

4.5 Manual Test Sensei

Soares et al. (2023a) proposed Manual Test Sensei, an NLP-

[9] Manual test case #1482 (hardware)

[10] Manual test case #1467 (hardware)

based tool to detect natural language test smells in manual test descriptions, including the smell types considered in this article. Their approach implements a set of hand-crafted identification rules using Python and spaCy (2026), relying primarily on lexical and shallow linguistic cues. In practice, each smell is associated with a list of indicator terms and patterns (e.g., specific keywords, verb forms, or markers of optionality and subjectivity), and the tool flags a step when these indicators are present.

In contrast, we adopt an LLM-based approach that supports richer semantic reasoning beyond surface-level keyword matching. Our work investigates a model-based setting in which Gemini-3-Pro-Preview performs smell detection by considering the full step text and its role in the test case structure, relying on explicit smell definitions and identification rules embedded in the prompt rather than fixed indicator lists. This design enables more context-sensitive judgments and can reduce false positives caused by purely lexical cues.

We used the results of Manual Test Sensei run on Ubuntu test cases (Soares et al., 2023a).[11] We evaluated 103 test steps sampled from 15 test cases, for which the first author had previously established a manual baseline. Using this baseline, Manual Test Sensei achieves 0.70 precision and 0.62 recall, with an F_1 score of 0.66. Overall, its performance is inferior to Gemini-3-Pro-Preview and ChatGPT 5.2 Thinking.

In some steps, both approaches agree. For example, in the Ubuntu test case `packages/1650_hardinfo`[12], the following action step appears:

```
<dt>Enter the name of the file and Click on
"Save" button</dt>
```

Both Manual Test Sensei and our Gemini-3-Pro-Preview-based approach correctly flag this step as Eager Action, since it bundles two distinct actions (entering the file name and clicking the "Save" button) into a single test step.

However, consider the test case `image/1337_Install (Default)`[13] and the following action step:

```
<dt>At "Write changes to disks", verify that
everything is right and select YES</dt>
```

The annotation produced by Gemini-3-Pro-Preview is as follows:

> ***Smells:*** *Ambiguous Test; Eager Action; Misplaced Verification*
> ***Explanation:*** *Contains indefinite pronoun 'everything' and subjective adjective 'right'. Combines multiple operations: 'verify' and 'select'. Uses verification verb 'verify' in the action field.*

Manual Test Sensei correctly detects Eager Action and Misplaced Verification. It also flags Unverified Action because there is no dedicated `<dd>` verification following this `<dt>`. In this particular case, however, the verification is embedded within the action step itself; thus, the absence of a separate `<dd>` is not necessarily incorrect, but rather reflects a redundant (or mixed) formulation. Finally, Manual Test Sensei does not identify Ambiguous Test in this step, whereas Gemini-3-Pro-Preview flags ambiguity due to the subjective expression "everything is right," which lacks an objective acceptance criterion.

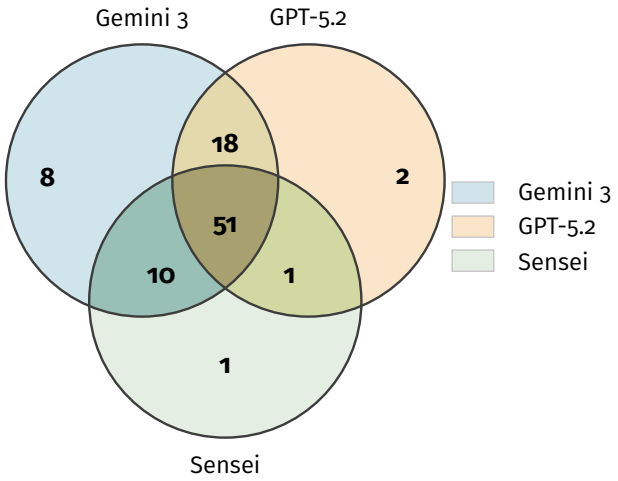

Figure 3. Venn diagram ($N = 103$) of steps where each tool exactly matches the baseline smell set.

Overall, Gemini-3-Pro-Preview attains the highest exact-match count (87/103): 51 steps are correctly labeled by all three tools, 18 are shared exclusively with ChatGPT 5.2 Thinking, 10 are shared exclusively with Sensei, and 8 are uniquely correct. ChatGPT 5.2 Thinking is correct in 72/103 steps (51 shared by all, 18 shared exclusively with Gemini-3-Pro-Preview, 1 shared exclusively with Sensei, and 2 uniquely correct). Sensei is correct in 63/103 steps (51 shared by all, 10 shared exclusively with Gemini-3-Pro-Preview, 1 shared exclusively with ChatGPT 5.2 Thinking, and 1 uniquely correct). Importantly, the tools are complementary: at least one tool matches the baseline in 91/103 steps, leaving 12 steps where none of the three tools reproduces the baseline exactly.

4.6 Threats to Validity

This study is subject to threats to validity that may affect our results and interpretations (Sallou et al., 2024). We discuss the main threats and the mitigation actions adopted.

4.6.1 Internal Validity

Our evaluation relies on manual annotations for a subset of the analyzed test cases. This process is inherently error-prone and may introduce subjectivity, since some smell definitions can leave room for interpretation and different annotators may disagree on borderline cases. To mitigate this threat, we base labeling on explicit definitions and identification rules, which are also embedded in the prompt, and we require the model to justify predictions by citing concrete textual triggers present in the step. In addition, the manually inspected sample was reviewed to reduce labeling mistakes and to avoid misinterpretations when assessing model outputs and explanations.

[11] `https://github.com/easy-software-ufal/manual-test-sensei/blob/main/ManualTestSensei/results-20230329-125316-en_core_web_lg.csv`
[12] Manual test case #1650 (packages)
[13] Manual test case #1337 (image)

4.6.2 Construct Validity

We operationalize smell detection as a multi-label classification problem at the step-level within a whole-test-case context. Although we analyze complete test cases, our ground truth is established by labeling individual steps according to the catalogued smells, and our metrics (precision, recall, and F_1 score) measure agreement with these labels rather than downstream impacts such as execution time, fault detection effectiveness, or maintenance effort. Moreover, explanation usefulness is assessed qualitatively by checking whether explanations cite evidence aligned with the smell definition, which remains a proxy for practical usefulness rather than a direct measure of developer benefit.

Another threat concerns the prompting strategy adopted in this study. The reported results depend on the specific combination of smell definitions, identification rules, examples, and meta-prompting instructions used to construct the final prompt. Alternative prompt formulations could lead to different performance levels. To facilitate replication and future comparisons, we provide the complete prompt used in our study. Investigating prompt sensitivity remains an important direction for future work.

4.6.3 External Validity

Our dataset is drawn from Ubuntu manual test cases, which follow a specific writing style and a structured `<dl>`/`<dt>`/`<dd>` representation. While Ubuntu is a widely used open-source system with a mature QA process, the prevalence of smells and the model performance may differ in other projects, domains, or manual testing formats. In addition, our evaluation is limited to a single contemporary model available at the time of the study (Gemini-3-Pro-Preview). Therefore, the results may not generalize to other LLMs or to smaller open models without further investigation.

Our results may also not fully generalize over time because LLM behavior can change due to model updates and API-level defaults, and outputs may vary under non-deterministic decoding. To support future replications, we report the execution period (January and February 2026) and adopt a fixed prompting strategy with strict output constraints. Even so, performance and explanations may differ in later versions of the model.

4.6.4 Conclusion Validity

Our conclusions about the effectiveness of Gemini-3-Pro-Preview are based on the test cases analyzed and on a manually inspected sample used to compute performance metrics. Although we observe frequent smell occurrences and encouraging detection performance, broader conclusions require additional evaluations on larger samples, more projects, and alternative test repositories. We therefore interpret the findings as evidence of practical promise rather than as definitive proof of general applicability.

5 Related Work

Hauptmann et al. (2013) argue that natural language test cases, which are widely used in system testing, are often written without following established best practices and therefore become difficult to maintain, hard to understand, and inefficient to execute. Inspired by the established notions of code smells and test smells, the authors introduce the concept of Natural Language Test Smells to characterize recurring quality problems in manual test cases written in natural language. To investigate the prevalence of these issues in practice, they conduct an empirical study on more than 2,800 test cases from seven industrial test suites, examining the occurrence of Natural Language Test Smells in real-world system-testing artifacts.

Soares et al. (2023a) present an empirical study on test smells in manual tests written in natural language, motivated by the observation that poorly described manual tests can suffer from issues analogous to those found in automated tests, potentially harming maintainability, coverage, and reliability. To address the limited knowledge about the types, frequency, and impact of such smells, the authors aim to contribute a dedicated catalog for manual test artifacts. They follow a two-fold strategy. First, they conduct an exploratory study on manual tests from three systems (Ubuntu, the Brazilian Electronic Voting Machine, and the user interface of a major smartphone manufacturer), and propose a catalog of eight natural language test smells along with identification rules based on syntactic and morphological text analysis, which they validate with 24 in-company test engineers. Second, they implement an NLP-based tool derived from these rules to automatically analyze the tests of the studied systems and validate its results. Their findings show frequent occurrences of the eight smells in practice; practitioners largely agree with the proposed definitions and examples (80.7% agreement), and the tool achieves high detection effectiveness (precision 92%, recall 95%, F_1 score 93.5%), reporting 13,169 smell occurrences across the analyzed systems. The study concludes that a dedicated catalog and NLP-based detection strategies can reduce the effort of analyzing manual tests, including in multilingual contexts.

Aranda et al. (2024) address the problem of test smells in natural language manual tests, noting that such smells can hinder testing activities by reducing maintainability, inducing non-deterministic behavior, and leading to incomplete verification. While prior work has catalogued smells in natural language tests, the authors highlight a gap in systematic and automated approaches for their removal. To fill this gap, they introduce (i) a catalog of transformations designed to remove seven natural language test smells and (ii) a companion tool that applies these transformations using Natural Language Processing techniques. Their goal is to improve the quality and reliability of natural language tests during software development. The paper evaluates these contributions through a two-fold empirical strategy. First, a survey with 15 software testing professionals assesses the acceptance and perceived usefulness of the proposed transformations. Second, an empirical study evaluates the automated tool by applying it to real-world test cases from the Ubuntu operating system. The results suggest that practitioners consider the

transformations valuable, and the tool achieves a promising level of effectiveness, reporting an F-measure of 83.70%.

Soares et al. (2025) investigate the impact of test smells in natural language manual test descriptions, a topic that has received substantially less attention than smells in automated tests. Motivated by the observation that such smells may harm maintainability, introduce non-deterministic execution, and lead to incomplete verification, the authors note that prior work has mostly focused on cataloging smells, with limited empirical evidence about their effects on test effectiveness. To address this gap, they conduct a controlled experiment with 30 participants from academia and industry and study two smells, Ambiguous Test and Eager Action. They evaluate whether these smells increase (i) test execution time, (ii) the number of steps or screens required to complete the tests (screen flow), and (iii) divergence in participants' perceptions of test success. Their results show that Ambiguous Test can increase execution time by up to five times and screen flow by up to seven times, whereas Eager Action does not increase execution time or screen flow when the grouped actions are dependent on one another. The study concludes by highlighting the need for better-designed manual test descriptions to improve clarity, consistency, and execution efficiency.

Beyond test cases, related efforts also use NLP to identify quality issues in other natural language artifacts. Veizaga et al. (2024), for example, proposed an automated approach to detect ambiguous or problematic linguistic patterns in requirements, and argued that contextualized rephrasing suggestions can assist analysts in improving specification quality.

In a similar spirit, Rajkovic and Enoiu (2022) developed NALABS to detect quality issues in requirements and test specifications written in natural language. Their approach relies on keyword-based indicators to compute metrics such as vagueness, referencability, optionality, subjectivity, and fragility, illustrating a representative class of rule-oriented methods for smell-like phenomena.

Other works have also explored test smells and related quality issues through the lens of language models. Yang et al. (2024) discussed challenges and systematic methods for exploring new smell types and combined multiple detection techniques to identify diverse smells. From a broader perspective,

Lucas et al. (2024) investigated the use of LLMs to detect a large set of test smells, emphasizing that these models can identify issues and often provide improvement suggestions. More generally, LLM-based quality analysis has been applied to other software artifacts as well.

Wu et al. (2024) proposed *iSMELL*, which integrates LLMs with specialized tools to detect and refactor code smells, dynamically selecting tools to improve scalability and performance. Although these studies focus on different artifacts and smell taxonomies, they collectively motivate the use of language models as flexible analyzers that can operate without extensive hand-crafted rule sets.

Bavota et al. (2015) conducted a controlled experiment with 20 master's students to investigate whether test smells hinder source code comprehension during software maintenance. The experiment involved program comprehension tasks performed on test suites with and without test smells, with participant performance measured in terms of correctness and time. The results indicate that several test smells have a strong negative effect on the understandability of both test suites and production code. In particular, test comprehension was found to improve by approximately 30% in the absence of test smells, reinforcing that these smells can substantially increase the cost of understanding and maintaining test artifacts.

In this article, we build on these foundations by evaluating Gemini-3-Pro-Preview for detecting natural language test smells in Ubuntu-style manual test cases. In contrast to prior rule-based or keyword-driven approaches, and extending our earlier investigation of SLMs (Lucas et al., 2025), we assess a contemporary LLM in a whole-test-case setting. Our approach leverages the `<dl>`/`<dt>`/`<dd>` structure to interpret action and verification roles, which is essential for role-dependent smells and for identifying unverified actions. We detected test smells in the 100 Ubuntu test cases analyzed, indicating that these issues occur frequently in practice. Moreover, in the manually inspected sample, Gemini-3-Pro-Preview achieved better performance than the SLMs evaluated in our prior work (Lucas et al., 2025), and produced concise, evidence-grounded explanations, suggesting that model-based detection can provide actionable support for practitioners aiming to improve the clarity and consistency of manual test cases.

6 Conclusions

In this article, we investigated the use of LLMs for detecting natural language test smells in Ubuntu-style manual test cases. We focused on a whole-test-case analysis setting, in which the model analyzes complete manual tests rather than isolated test-case steps. This setting allows the model to exploit contextual information across actions and verifications and better reflects how practitioners inspect manual testing artifacts. We operationalized seven smell types through a structured prompting strategy that leverages the `<dl>` organization of Ubuntu tests, and we showed that model-generated explanations can support the identification of ambiguous, complex, misplaced, or unverified steps. Overall, our results provide evidence that LLM-based detection can scale test smell analysis beyond rule-based techniques and support quality assessment of manual testing artifacts.

Our findings also highlight clear benefits when moving from SLMs to a contemporary LLM. In particular, Gemini-3-Pro-Preview consistently outperformed the SLMs evaluated in our earlier study (Lucas et al., 2025), while producing more actionable explanations that can help practitioners revise test steps for greater clarity and consistency. This performance gap suggests that, at least for this task and dataset, stronger models can better capture subtle linguistic cues and contextual dependencies that are common in manual test descriptions.

At the same time, deploying model-based smell detection in real workflows raises challenges. Inference cost and latency may be nontrivial for large test suites, and model outputs can vary across prompts, decoding settings, and model

updates. Moreover, incorrect classifications may introduce noise in review processes if not accompanied by conservative thresholds and human oversight. Despite these constraints, our results are promising and suggest that language models can provide practical assistance for improving manual tests, especially when the goal is to support reviewers and testers rather than to fully replace human judgment.

First, we plan to expand the evaluation to substantially more test cases and to additional manual-testing repositories beyond Ubuntu, including test suites written in other languages, to assess generalization across writing styles, domains, and linguistic contexts. Given the strong performance of current models on our English dataset, we expect this direction to be feasible and potentially effective, particularly for widely supported languages. Second, we will consider more projects and larger, more diverse test suites to better characterize how smell prevalence and detection performance vary across contexts. Third, we will incorporate agent-based workflows, following recent approaches in the literature Melo et al. (2026), to iteratively inspect test cases, cross-check step-level predictions with test-case-level context, and reconcile borderline cases (e.g., ambiguity versus acceptable flexibility). Fourth, beyond accuracy-oriented measures, we will evaluate reliability and stability to quantify the consistency of model outputs across repeated runs. Specifically, we will adopt a set of accuracy and stability metrics inspired by prior work on LLM evaluation (Chen et al., 2021; Atil et al., 2025), which allow us to assess whether a model produces correct results and whether it does so consistently across multiple attempts—an essential property for practical developer-facing tools. Fifth, and most importantly, we will investigate to what extent models can *refactor* natural language manual tests to remove these smells, building on our prior experience with refactoring automated test cases to eliminate test smells (Melo et al., 2026; Soares et al., 2023b, 2020).

Concretely, we will evaluate model-assisted editing suggestions that (i) split coarse steps to mitigate *Eager Action*, (ii) rewrite vague statements into objectively checkable criteria to mitigate *Ambiguous Test*, (iii) relocate checks between action and verification fields to mitigate *Misplaced Action* and *Misplaced Verification*, (iv) make conditions explicit and structured to mitigate *Conditional Test*, and (v) add missing expected outcomes to mitigate *Unverified Action*. We will measure not only smell reduction (before/after) but also semantic preservation of the intended procedure, readability, and the amount of human effort required to accept or revise the proposed edits. Finally, we aim to evaluate additional model families and sizes and to explore fine-tuning strategies that distill the performance of proprietary models into smaller and lighter open models, enabling lower-cost and privacy-preserving deployments without sacrificing accuracy. Together, these directions can further clarify the practical boundaries and opportunities for scalable, model-assisted quality assurance and refactoring of natural language manual tests.

Declarations

Acknowledgements

We thank the anonymous reviewers for their constructive feedback and insightful suggestions, which significantly improved the quality and clarity of this article.

Funding

This work was partially supported by CNPq (306026/2026-0, 408040/2025-4, 403719/2024-0), CAPES (88887.313474/2026-00), FAPESQ-PB (268/2025), and FAPEAL grants. This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) `http://www.ines.org.br`, CNPq grant 408817/2024-0.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

All artifacts are publicly available online (Lucas et al., 2026).

References

Abdin, M., Aneja, J., Behl, H., Bubeck, S., Eldan, R., Gunasekar, S., Harrison, M., Hewett, R. J., Javaheripi, M., Kauffmann, P., et al. (2024). Phi-4 technical report. *arXiv preprint arXiv:2412.08905*.

Aranda, M., Oliveira, N., Soares, E., Ribeiro, M., Romão, D., Patriota, U., Gheyi, R., Souza, E., and Machado, I. (2024). A catalog of transformations to remove smells from natural language tests. In *International Conference on Evaluation and Assessment in Software Engineering*, pages 7–16. ACM.

Atil, B., Aykent, S., Chittams, A., Fu, L., Passonneau, R. J., Radcliffe, E., Rajagopal, G. R., Sloan, A., Tudrej, T., Ture, F., Wu, Z., Xu, L., and Baldwin, B. (2025). Non-determinism of "deterministic" LLM settings.

Basili, V. R., Caldiera, G., and Rombach, H. D. (1994). *The Goal Question Metric Approach*.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., and Binkley, D. (2015). Are test smells really harmful? An empirical study. *Empirical Software Engineering*, 20(4):1052–1094.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners. In *Advances in Neural Information Processing Systems*.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman,

G., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chiang, W.-L., Zheng, L., Sheng, Y., Angelopoulos, A. N., Li, T., Li, D., Zhu, B., Zhang, H., Jordan, M. I., Gonzalez, J. E., and Stoica, I. (2024). Chatbot arena: an open platform for evaluating LLMs by human preference. In *International Conference on Machine Learning*, ICML'24. JMLR.org.

DAIR.AI (2024). Prompt Engineering Guide. `https://www.promptingguide.ai/techniques`.

Hauptmann, B. (2016). *Reducing system testing effort by focusing on commonalities in test procedures*. PhD thesis, Technische Universität München.

Hauptmann, B., Junker, M., Eder, S., Heinemann, L., Vaas, R., and Braun, P. (2013). Hunting for smells in natural language tests. In *International Conference on Software Engineering*, pages 1217–1220. IEEE Computer Society.

Hou, Y., Dong, H., Wang, X., Li, B., and Che, W. (2022). Metaprompting: Learning to learn better prompts. *arXiv preprint arXiv:2209.11486*.

Juhnke, K., Nikic, A., and Tichy, M. (2021). Clustering natural language test case instructions as input for deriving automotive testing DSLs. *J. Object Technol.*, 20(3):5–1.

Junior, N. S., Martins, L., Rocha, L., Costa, H., and Machado, I. (2021). How are test smells treated in the wild? A tale of two empirical studies. *Journal of Software Engineering Research and Development*, 9:9–1.

Junior, N. S., Rocha, L., Martins, L. A., and Machado, I. (2020). A survey on test practitioners' awareness of test smells. In *Iberoamerican Conference on Software Engineering*, pages 462–475. Curran Associates.

Lucas, K., Gheyi, R., Ribeiro, M., Palomba, F., Martins, L., and Soares, E. (2025). Investigating the performance of small language models in detecting test smells in manual test cases. In *Proceedings of the Brazilian Symposium on Software Engineering*, pages 783–789, Porto Alegre, RS, Brasil. SBC.

Lucas, K., Gheyi, R., Ribeiro, M., Palomba, F., Martins, L., and Soares, E. (2026). An empirical study of Gemini 3 for detecting natural language test smells in manual test cases (artifacts). `https://doi.org/10.5281/zenodo.20549059`.

Lucas, K., Gheyi, R., Soares, E., Ribeiro, M., and Machado, I. (2024). Evaluating large language models in detecting test smells. In *Brazilian Symposium on Software Engineering*, pages 672–678.

Melo, R., Simoes, P., Gheyi, R., d'Amorim, M., Ribeiro, M., Soares, G., Almeida, E., and Soares, E. (2026). Agentic LMs: Hunting Down Test Smells. *IEEE Software*.

Peixoto, M., Baia, D., Nascimento, N., Alencar, P., Fonseca, B., and Ribeiro, M. (2025). On the Effectiveness of LLMs for Manual Test Verifications . In *2025 IEEE/ACM International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)*, pages 45–52, Los Alamitos, CA, USA. IEEE Computer Society.

Rajkovic, K. and Enoiu, E. (2022). NALABS: Detecting bad smells in natural language requirements and test specifications. *arXiv preprint arXiv:2202.05641*.

Sallou, J., Durieux, T., and Panichella, A. (2024). Breaking the silence: the threats of using LLMs in software engineering. In *International Conference on Software Engineering: New Ideas and Emerging Results*, pages 102–106. ACM.

Soares, E., Aranda, M., Oliveira, N., Ribeiro, M., Gheyi, R., Souza, E., Machado, I., Santos, A. L. M., Fonseca, B., and Bonifácio, R. (2023a). Manual tests do smell! Cataloging and identifying natural language test smells. In *International Symposium on Empirical Software Engineering and Measurement*, pages 1–11. IEEE.

Soares, E., Ribeiro, M., Amaral, G., Gheyi, R., Fernandes, L., Garcia, A., Fonseca, B., and Santos, A. (2020). Refactoring test smells: A perspective from open-source developers. In *Brazilian Symposium on Systematic and Automated Software Testing*, pages 50–59.

Soares, E., Ribeiro, M., Gheyi, R., Amaral, G., and Santos, A. L. M. (2023b). Refactoring test smells with JUnit 5: Why should developers keep up-to-date? *IEEE Transactions on Software Engineering*, 49(3):1152–1170.

Soares, G., Santos, V., Ribeiro, M., Martins, L., Pontillo, V., III, M. A., Gheyi, R., Machado, I., and Palomba, F. (2025). On the harmfulness of test smells in manual system testing: A controlled experiment. In *International Symposium on Empirical Software Engineering and Measurement*. IEEE.

spaCy (2026). Industrial-strength natural language processing in Python. `https://spacy.io/`.

Team, G., Kamath, A., Ferret, J., Pathak, S., Vieillard, N., Merhej, R., Perrin, S., Matejovicova, T., Ramé, A., Rivière, M., et al. (2025). Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. (2023). LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Ubuntu (2026). Manual tests. `https://launchpad.net/ubuntu-manual-tests`.

Veizaga, A., Shin, S. Y., and Briand, L. C. (2024). Automated smell detection and recommendation in natural language requirements. *IEEE Transactions on Software Engineering*, 50(4):695–720.

Wu, D., Mu, F., Shi, L., Guo, Z., Liu, K., Zhuang, W., Zhong, Y., and Zhang, L. (2024). iSMELL: Assembling LLMs with Expert Toolsets for Code Smell Detection and Refactoring. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1345–1357.

Yang, Y., Hu, X., Xia, X., and Yang, X. (2024). The lost world: Characterizing and detecting undiscovered test smells. *ACM Transactions on Software Engineering and Methodology*, 33(3).