NoseSense: Benchmarking Tool

Magno Macedo
Computer Institute, Federal
University of Bahia
Salvador, Brazil
magnomiranda@ufba.br

Lucas Fraga
Computer Institute, Federal
University of Bahia
Salvador, Brazil
lucasfraga@ufba.br

Letícia Barreto
Computer Institute, Federal
University of Bahia
Salvador, Brazil
leticiabarreto@ufba.br

Tássio Virginio
Computer Institute, Federal
University of Bahia
Salvador, Brazil
tassio.virginio@ufba.br

Ivan Machado
Computer Institute, Federal
University of Bahia
Salvador, Brazil
ivan.machado@ufba.br

Abstract

The rapid evolution of Large Language Models (LLMs), characterized by a shift from yearly to weekly advancements, has rendered traditional static evaluation studies quickly obsolete upon publication. In this fast-paced landscape, evaluation methodologies must be designed to enable immediate and continuous assessment of newly released models. Automated benchmarking has emerged as a practical solution, allowing LLMs to be evaluated and compared as soon as they become available. A wide range of benchmarks has been proposed to assess LLM capabilities across diverse application domains. However, there is a lack of targeted evaluation frameworks focusing on software engineering tasks related to test quality. In this context, we present an automated benchmarking tool designed to evaluate the accuracy of LLMs in identifying anti-patterns in test code, commonly referred to as Test Smells. Our approach enables scalable, reproducible, automated, and up-to-date assessment of LLM performance in this critical aspect of software quality. The tool demonstration video can be found on zenodo https://zenodo.org/records/20277258 and youtube https://youtu.be/D0o77YuM_Xs.

Keywords

LLM, Benchmarks, Tools, Test Smells

1 Introduction

The vast presence of digital systems in modern society has established robustness and reliability as fundamental pillars of Software Engineering. In this context, maintaining high-quality test suites is imperative to mitigate failures and reduce long term operational costs. However, the effectiveness of these tests is frequently compromised by design anti-patterns called test smells that do not impair code functionality, but degrade its readability, maintainability, and evolution [14].

The search for automated solutions to identify and refactor these anti-patterns has found a promising yet complex path in Large Language Models (LLMs) [2, 5, 20]. This complexity stems primarily from a critical methodological challenge: the temporal volatility of experimental results. The innovation cycle of LLMs has been so accelerated that comparative studies of benchmarks tend to become saturated and obsolete even before their publication [1, 7]. In the interval required to conclude a rigorous investigation, the release of new model iterations often invalidates static benchmarks, turning the state of the art into a moving target, which undermines the ability to use the study's findings as a foundation for further research or industrial application.

To mitigate this issue, we present NoseSense, a dynamic benchmarking tool designed to simultaneously evaluate the performance of multiple LLMs in detecting test smells. To enable scalable and reproducible evaluation, the tool operationalizes this detection task through a structured multiple-choice benchmark: each sample presents a code snippet containing a test smell, and the model must identify which smell is present by selecting among a set of predefined options. This controlled format ensures that model responses can be automatically verified against a ground-truth oracle, allowing rigorous measurement of each LLM's ability to recognize test smell patterns.

Our project provides a robust solution that addresses the limitations of traditional studies on this topic, which often become obsolete during the lengthy writing and publication process. Instead of time-consuming longitudinal analyses, the project prioritizes rapid, practical validation, enabling real-time comparisons of the most current models. The necessity for tools like NoseSense is reinforced by the high prevalence of anti-patterns in real-world systems [2], empirical studies indicate that between 18% to 26% of test suites are free of test smells [4, 16]. Thus, NoseSense serves as a diagnostic instrument for the semantic classification capability of LLMs with respect to test code quality, providing continuously updated metrics that keep pace with the industry's rapid technological evolution.

2 Test Smells

Van Deursen et al. [13] argue that test smells possess their own taxonomy and corresponding remedies, identifying eleven distinct types in their seminal work. Later studies expanded this taxonomy by proposing additional smell types and examining their distribution and co-occurrence patterns [3, 8, 10, 13–15]. To evaluate the effectiveness of NoseSense, we selected a set of 10 test smells, detailed below:

Assertion Roulette (AR): Occurs when a test method encapsulates multiple assertions without explanatory messages.

Conditional Test Logic (CTL): Characterized by the presence of control structures in the test body.

Duplicate Assert (DA): Manifests through the repetition of assertions that validate identical states or conditions within the same test scope.

Eager Test (ET): Occurs when a single test method attempts to verify multiple functionalities or production class methods simultaneously.

Exception Catching Throwing (ECT): Refers to the improper use of `try-catch` blocks to catch exceptions that should be propagated to the test framework.

Ignored Test (IT): Identifies tests that were explicitly disabled (e.g., `@Ignore` annotation).

Magic Number (MN): Consists of using literal numbers without explicit semantic context in assertions.

Sensitive Equality (SE): Occurs when the equality criterion relies on the textual representation (`toString`) of complex objects.

Unknown Test (UT): Represents methods that execute production code but lack assertions.

Verbose Test (VT): Characterized by excessively long test methods.

3 NoseSense

NoseSense provides a controlled and systematic environment for evaluating the capability of LLMs in detecting test smells. To ensure the statistical rigor of the analyses, the tool integrates an oracle composed of 1,000 code instances, comprising 100 samples for each of the 10 test smells selected and described in Section 2. This oracle is used to assess the precision and diagnostic accuracy of the evaluated artificial intelligence models. The following subsections detail the tool's features, systemic architecture, and operational workflow. The oracle dataset focuses on Java test code and it happens for two main reasons: first, this language remains the most widely studied in the test smell literature [8, 13–15]; second, JNose, the static analysis tool utilized to extract and validate the ground-truth samples for our oracle, was exclusively designed to operate on Java projects. This methodological choice ensures compatibility with established definitions and provides robust empirical foundations.

3.1 Features

NoseSense was designed with a set of features that cover the entire model evaluation lifecycle, from configuration to results analysis:

Provider and Model Registration: Enables the inclusion of multiple AI platforms with their credentials, allowing quick adoption of newly released models.

Automated Benchmark Execution: Evaluates multiple LLMs simultaneously against an oracle of 1,000 test smell samples, with real-time monitoring and interruption capabilities.

Analytical Dashboard: Consolidates results into graphical visualizations (bar, pie, and radar charts) to easily compare overall accuracy and specific capabilities.

Results Persistence and Export: Automatically stores evaluation history in a relational database and allows full CSV exports to ensure experiment reproducibility.

3.2 System Architecture

NoseSense adopts a client-server architecture based on the decoupling of a backend and a reactive user interface. This layered organization ensures that the model orchestration logic operates independently from the presentation layer, promoting high maintainability, simplicity and ecosystem scalability [9].

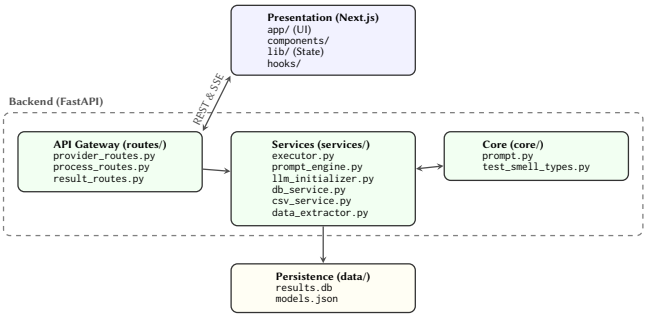

Figure 1: NoseSense System Architecture Diagram.

As detailed in Figure 1, the system's core, located in the `/backend` directory, utilizes the `FastAPI` framework[1] , selected for its asynchronous nature. The structure is subdivided into logical layers ranging from data reception to persistence. Interaction begins at the Routes Layer (API Gateway), situated in `routes/`, which acts as the entry point and isolates responsibilities into specific domains: provider management (`provider_routes.py`), analysis execution (`process_routes.py`), and history retrieval (`result_routes.py`).

Below this interface, the system's business rules reside in the Services Layer (`services/`). At this level, the `executor.py` component manages call concurrency to avoid rate limit penalties, while `prompt_engine.py` applies randomization rules to the multiple-choice options to ensure the randomness of the tests. Instance initialization, performed via LangChain, is isolated in the `llm_initializer.py` module, ensuring the modularity of the inference engine. Complementarily, the Core Layer encapsulates the essential definitions of test smells in the `core/` directory, centralizing the application's domain.

Finally, information management occurs in the Persistence Layer (`data/`), which adopts a hybrid approach: configuration metadata and credentials are kept in `JSON` files for flexibility, while evaluation results are persisted in a relational database. This strategy ensures referential data integrity, facilitates metric extraction, and enables automated export through the `csv_service.py` service.

Complementing the backend, the system's presentation layer (`/frontend`) is built with `Next.js`[2] and `React`[3] technologies, aiming to provide a fluid and data-driven User Experience. The structuring of pages and visual components (`app/`) employs the App Router paradigm to manage navigation and global layout, significantly optimizing rendering.

To ensure the cohesion of the ecosystem illustrated in Figure 1, the communication between frontend and backend utilizes the `REST` approach for static configurations and registrations. However, for

[1] https://fastapi.tiangolo.com/
[2] https://nextjs.org/
[3] https://react.dev/

the critical and prolonged process of test smell analysis, the system employs the Server-Sent Events (SSE) protocol.

3.3 Environment Initialization and Configuration

The application's lifecycle begins with the bootstrapping of the `FastAPI` server. In this phase, the system establishes communication protocols via `CORS` middleware and maps the API routes. Concurrently, the user submits access credentials and AI provider metadata through the interface; this information is then processed by the backend and stored in a structured manner in a local `JSON` repository for use in testing sessions, as illustrated in the configuration form in Figure 3.

3.4 Experiment Preparation

Upon starting a new round of analyses, NoseSense establishes a Server-Sent Events (SSE) connection to provide real-time monitoring, triggering an orchestration function responsible for the preparatory procedures. Initially, the system performs the dynamic instantiation of the selected models, a process that involves resolving API keys and configuring essential technical parameters for inference. Subsequently, the extraction of the dataset occurs, in which code snippets containing test smells are retrieved from an experimental corpus of 1,000 samples. These data are stored in the `SQLite` database. This step is essential for individually monitoring the status and performance of each task, enabling the subsequent retrieval of results and execution-time fault auditing.

3.5 Prompt Engineering and Standardization

The validity of a benchmark involving LLMs depends on the quality and consistency of the instructions provided. To ensure replicability and align prompt construction with industry practices, the prompt engine of NoseSense is designed based on the systematic study of prompt templates in real-world LLM-based applications (LLMapps) conducted by Mao et al. [6]. We adopted the CRISPE framework (Capacity and Role, Insight, Statement, Personality, and Experiment) [17] to structure instructions, reducing operational instability and establishing a deterministic classification task. The mapping and implementation of these components within NoseSense are detailed below:

Capacity and Role (CR) – Persona Definition:
Implementation: "You are a senior software engineer..."
Justification: We chose the persona of a senior software engineer to direct the LLMs toward rigorous code quality criteria, mitigating naive interpretations.

Insight (I) – Contextualization (Few-Shot/ICL):
Implementation: Detailed definitions of the 10 selected test smells followed by Java code examples.

Statement (S) – Execution Directive:
Implementation: "In the test code below..." (as exemplified in Figure 4).
Justification: The use of this directive to convert the open-ended reasoning task into a closed, structured classification by restraining the scope of the question to the relevant part.

Personality (P) – Format Restriction:
Implementation: "Answer format: {{A}}"
Justification: By limiting the format of the question, we ensure the LLMs will not give open ended answers to help with the predictability and precise automated extraction of responses for a large-scale benchmark.

Experiment (E) – Optional Recapitulation:
Implementation: Excluded based on task specificity.
Justification: This aspect was not used in the prompt due to lack of relevancy to the task.

3.6 Evaluation Cycle and Concurrent Execution

For each identified code artifact, the system initiates an iterative evaluation sub-process that integrates everything from input refinement to data consolidation. The cycle begins in the prompt engineering phase, where the engine contextualizes the code snippet and applies randomization techniques to the 4 multiple-choice options to neutralize potential position biases, invariably keeping the fifth alternative as "None". Figure 4 illustrates a simplified instance of the prompt submitted to the models.

The execution of tasks with the LLMs occurs asynchronously, under a semaphore-based concurrency control mechanism. This strategy allows limiting the volume of simultaneous calls, mitigating the risk of rate limiting blockages from providers. During the AI processing period, the backend emits keep-alive signals to prevent premature termination of the SSE data flow and ensure communication stability. Real-time monitoring of this processing is visible in the system interface (Figure 5).

Upon reception the system parses the responses to extract the generated diagnosis. This information is automatically compared with the dataset's expected ground truth, with the final result and execution metadata persisted in the relational database for subsequent analyses. Figure 6 demonstrates the immediate comparison interface between the AI diagnosis and the correct answer. Whenever one of the LLMs provides an incorrect classification, the interface displays not only the selected alternative letter but also resolves and presents the name of the test smell that the model erroneously identified. This diagnostic enrichment facilitates rapid qualitative analysis of misclassification patterns without requiring the researcher to manually cross-reference randomized option mappings.

3.7 Results and Artifact Consolidation

Upon completion of all iterations, the system terminates the processing pipeline. The raw data stored in `SQLite` is processed to generate a consolidated report in `CSV` format, providing a macroscopic view of each model's accuracy. The workflow finishes with the sending of a completion event, signaling the end of the experiment to the user interface.

3.8 Data Consolidation and Visualization

Following the analysis pipeline, NoseSense generates an interactive dashboard that transforms raw data into structured visualizations. The initial interface presents a statistical summary with four macro-indicators (total tests, correct responses, classification errors, and API failures) and, for rapid interpretation, highlights the overall

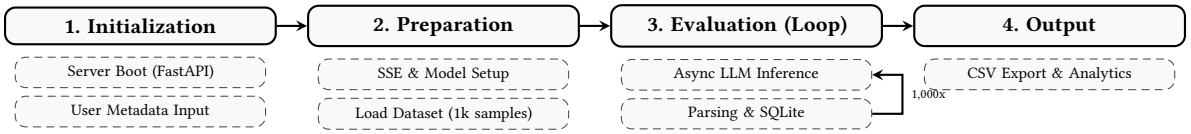

Figure 2: Simplified NoseSense Workflow Architecture.

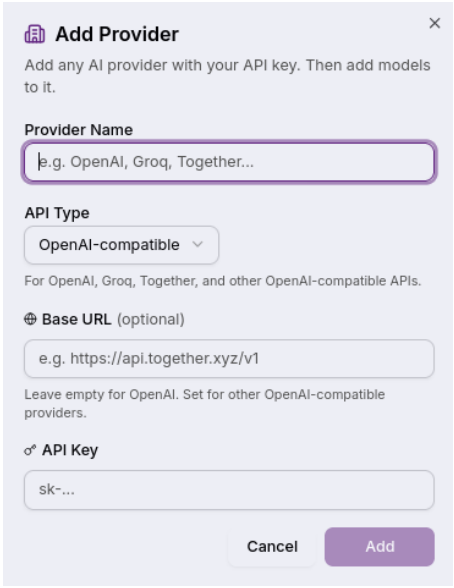

Figure 3: Providers Input Form.

Context: "Test smells" are antipatterns in unit test code indicating potential design problems [...]

Definitions:
Duplicate Assert: occurs when a test method tests for the same condition multiple times [...]
Assertion Roulette: occurs when a test method has multiple non-documented assertions [...]
(+ 8 additional smell definitions with examples)

Role: You are a senior software engineer.

Question: In the test code below, which type of "Test Smell" can be identified?

```
@Test
public void testExample() {
 assertEquals(42, calc.sum(40, 2));
 assertEquals(42, calc.sum(40, 2));
}
```

Alternatives:
A: Eager Test
B: Duplicate Assert
C: Verbose Test
D: Ignored Test
E: None

Directive: Provide only one alternative letter.
Answer format: {B}

Figure 4: Example of a prompt submitted to the LLMs

accuracy, which reached 43.7% in preliminary experiments , alongside a comparison of the best and worst-performing models in the round.

Analytical depth is reinforced by comparative bar charts that contrast the volume of correct answers, errors, and API failures, segmented by both model and test smell category. This analysis is complemented by a radar chart displaying the aggregated accuracy for the ten anomaly categories, which allows for the identification

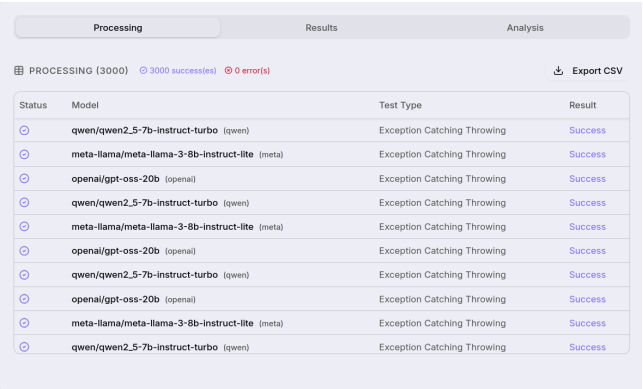

Figure 5: Real-time processing of executed tests and models.

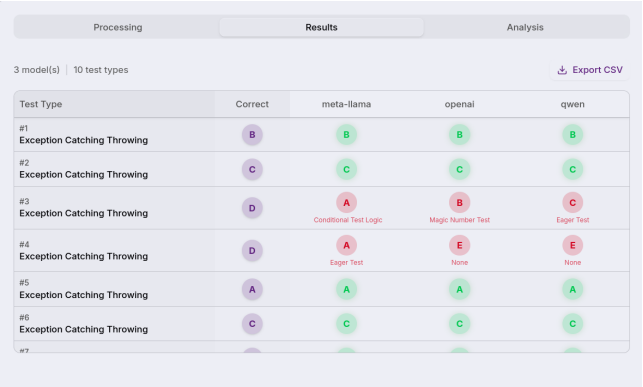

Figure 6: Real-time response analysis comparing the LLM response with the correct answer.

of the AIs' semantic understanding patterns, while a pie chart summarizes the experiment's integrity by illustrating the global proportion between correct and incorrect answers.

The final data organization ranks accuracy rates, detailing individual model and test category performance through chromatic indicators and progress bars. The dashboard consolidates a rigorous statistical summary, including average accuracy, standard deviation, and residual API error rates, ensuring the interface provides scientific groundwork for auditing and evolving the evaluated models, as exemplified in Figure 7.

4 Experimentation

As a proof of concept, we evaluated three distinct models. The selection criteria prioritized three axes of diversity: (i) provider heterogeneity, covering distinct API ecosystems (OpenAI, Meta, and

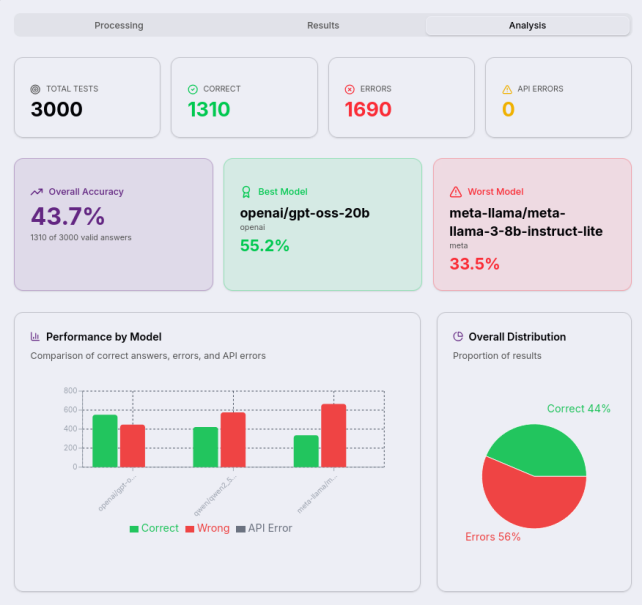

Figure 7: Selected charts showing the final analysis.

Alibaba); (ii) scale variation, ranging from 7B to 20B parameters, to assess whether model size correlates with detection accuracy; and (iii) licensing diversity, contrasting proprietary and open-source architectures. It is important to note that this selection is illustrative rather than exhaustive; NoseSense allows new models to be registered and evaluated as they become available, without changes to the evaluation pipeline. All models were accessed remotely via the Together AI API[4]. Table 1 details the selected models.

Table 1: Language Models Evaluated in the Experiment

Provider	Specific Models	Motivation
OpenAI	GPT OSS 20B	Large scale and proprietary API
Meta	Llama 3 8B	Open-source efficiency
Qwen	Qwen 2.5 7B	Instruction optimization

4.1 Functional Proof via Metrics

The results obtained during the benchmark execution provide a quantitative view of LLM performance against test smells. Figure 8 presents the overall accuracy ranking, while Table 2 details the segmented performance by anomaly category.

The tool's efficacy in providing diagnostics subsidies for LLMs is proven by analyzing the generated artifacts. Through the Detectability Ranking, the system accurately identified which anomalies are more "visible" to current semantic understanding; the Exception Catching Throwing category, for example, showed high detectability (87.3%), contrasting with Ignored Test, which emerged as the greatest challenge for the models (14.7%).

Regarding Performance Discrimination, NoseSense was able to rank the models by accuracy, pointing to GPT OSS 20B as the most effective on the proposed dataset (55.2%) and Llama 3 8B as the

[4]https://www.together.ai/

Table 2: LLM Accuracy by Test Smell Category

Test Smell Category	Correct/Total	Accuracy
Exception Catching Throwing	262/300	87.3%
Duplicate Assert	199/300	66.3%
Verbose Test	167/300	55.7%
Assertion Roulette	133/300	44.3%
Magic Number Test	122/300	40.7%
Conditional Test Logic	117/300	39.0%
Sensitive Equality	108/300	36.0%
Eager Test	89/300	29.7%
Unknown Test	69/300	23.0%
Ignored Test	44/300	14.7%

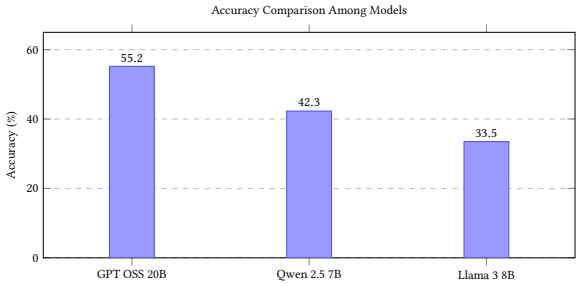

Figure 8: Accuracy results obtained.

least accurate (33.5%). Finally, the platform consolidated comprehensive statistical reports, including a 43.7% average accuracy and a 8.9% standard deviation among the models, allowing for immediate scientific data interpretation after test completion.

4.2 Multiclass Confusion Analysis

Beyond aggregate accuracy, a deeper understanding of model behavior requires analyzing how misclassifications are distributed across categories. To this end, NoseSense generates a multiclass confusion matrix that cross-references actual test smell labels with the classifications produced by the LLMs. Since the multiple-choice options are randomized per sample, the system resolves each model response back to the corresponding smell name before aggregation. Table 3 presents the aggregated confusion matrix for the three evaluated models.

Table 3: Aggregated Multiclass Confusion Matrix

	AR	CTL	DA	ET	ECT	IT	MN	SE	UT	VT	None
AR	**133**	6	24	26	13	7	22	8	4	6	51
CTL	12	**117**	28	20	26	5	16	4	7	26	39
DA	14	8	**199**	17	9	3	10	2	3	13	22
ET	4	2	10	**89**	26	1	17	5	8	14	124
ECT	1	3	3	4	**262**	1	1	0	3	0	22
IT	16	7	18	26	35	**44**	21	4	11	21	97
MN	9	2	17	24	23	4	**122**	6	5	16	72
SE	14	7	14	9	25	3	20	**108**	5	5	90
UT	10	10	14	26	24	8	19	6	**69**	5	109
VT	12	3	13	23	17	8	17	3	6	**167**	31

The confusion matrix reveals several notable misclassification patterns. The "None" column is particularly informative: categories such as Eager Test (124), Unknown Test (109), Ignored Test (97), and Sensitive Equality (90) exhibit high rates of "None" selections,

suggesting that the models frequently fail to recognize these anti-patterns entirely rather than confusing them with other smells. In contrast, Exception Catching Throwing shows minimal "None" selections (22), consistent with its high overall accuracy.

To provide a more rigorous evaluation beyond simple accuracy, Table 4 presents the per-class Precision, Recall, and F1-Score metrics derived from the confusion matrix. Precision measures the proportion of correct predictions among all samples classified as a given category, while Recall corresponds to the proportion of actual instances that were correctly identified.

Table 4: Per-class Accuracy, Precision, Recall, and F1-Score

Test Smell	Acc.	Prec.	Rec.	F1
ECT	92.1%	57.0%	87.3%	68.9%
DA	91.9%	58.5%	66.3%	62.2%
VT	92.0%	61.2%	55.7%	58.3%
AR	91.4%	59.1%	44.3%	50.7%
CTL	92.3%	70.9%	39.0%	50.3%
SE	92.3%	74.0%	36.0%	48.4%
MN	89.3%	46.0%	40.7%	43.2%
ET	87.1%	33.7%	29.7%	31.6%
UT	90.6%	57.0%	23.0%	32.8%
IT	90.1%	52.4%	14.7%	22.9%
Macro Avg.	**90.9%**	**57.0%**	**43.7%**	**46.9%**

Per-category analysis reveals an asymmetry between Precision and Recall: the Sensitive Equality and Conditional Test Logic classes achieve high precision (reliable predictions) but low recall (many instances go undetected). The opposite occurs with Exception Catching Throwing, which presents high recall (few omissions) but moderate precision due to false positives. The macro F1-Score of 46.9% confirms that while there is reasonable detection capability in some scenarios, there remains significant room for improvement, especially for Ignored Test and Eager Test. These results reinforce the role of NoseSense as a diagnostic instrument, as the confusion analysis maps the semantic boundaries where LLMs fail, offering actionable insights for prompt engineering refinement and model selection.

5 Related Works

The rapid advancement of LLMs has driven the development of benchmarking tools, such as promptfoo, which, like NoseSense, enables the comparison of multiple models. They share a focus on local and integrated execution, offering an automated environment for validation and analytical report generation [11, 12, 18].

The main divergence between the two tools lies in specialization: promptfoo has a broad, general-purpose focus, whereas NoseSense is highly specific to test smell detection. NoseSense is based on a validated 1,000-sample oracle labeled by JNose [15], a structured CRISPE-based prompt engine tailored to test smell taxonomy, and a custom analytical dashboard that produces domain-relevant metrics, per-category accuracy, multiclass confusion matrices, alongside per-class Precision, Recall, and F1-Score, which promptfoo does not natively provide [11, 12, 18].

Furthermore, NoseSense integrates real-time SSE-based monitoring specifically designed for long-running LLM inference sessions, whereas promptfoo targets shorter prompt evaluation cycles. In summary, NoseSense does not compete with promptfoo as a general benchmarking framework; rather, it delivers a ready-to-use, zero-configuration instrument that allows researchers to immediately evaluate any newly released model against an established software quality oracle, without requiring expertise in prompt engineering or dataset curation [11, 12, 18].

6 Threats to Validity

We identified several threats to the validity of our study, organized according to established categories [19].

Construct Validity: Our prompt engineering strategy utilized a single template style, CRISPE, which may limit the generalizability of our conclusions. The multiple-choice format restricts the models' response space and may introduce bias, as models are guided to choose from predefined alternatives.

Internal Validity: Our metrics are based on a single execution per sample for each model. Since LLMs are inherently non-deterministic, repeated executions of the same prompt may yield different responses. The absence of multiple runs limits our ability to measure response variance. Additionally, the use of default hyperparameters for model inference may influence the determinism of the generated responses.

External Validity: Due to our choice of Java, our data may not generalize to other programming languages. The oracle dataset is subject to the constraints of the JNose tool. Furthermore, the evaluation dataset represents only a limited sample of the real world, and results may vary across other project domains.

Conclusion Validity: No qualitative study was conducted to evaluate the tool with end-users to verify reduction of effort, results, usability metrics, or evaluations based on real-world tasks.

7 Conclusion

This paper presented NoseSense, a dynamic benchmarking tool designed to address the rapid obsolescence of traditional static studies evaluating LLMs. By shifting from time-consuming, static analyses to a pragmatic, real-time validation platform, NoseSense provides a continuous and reliable metrology instrument for assessing AI efficacy in test smell detection.

Through its layered architecture and the use of low-latency communication protocols like SSE, the tool efficiently orchestrates the simultaneous execution of heterogeneous LLMs. Our empirical demonstration, encompassing an automated evaluation pipeline and a comprehensive data visualization environment for 10 distinct test smell categories, proves NoseSense's utility in simplifying the interpretation of complex benchmarking results. NoseSense helps reduce the barrier for researchers and engineers to make empirically informed decisions about the selection of the model for software quality assurance.

Artifact Availability

The NoseSense tool, including its source code, oracle dataset, and replication package, is publicly available at: https://doi.org/10.5281/zenodo.22149462 and https://github.com/arieslab/NoseSense under the MIT License.

References

[1] Ryan T Allen and Rory M McDonald. 2026. How well can AI do strategy? Empirical benchmarking using strategy simulations. *Strategy Science* 11, 1 (2026), 93–117.

[2] Victor Alves, Carla Bezerra, Ivan Machado, Larissa Rocha, Tassio Virgínio, and Publio Silva. 2025. Quality Assessment of Python Tests Generated by Large Language Models. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering (EASE '25)*. Association for Computing Machinery, New York, NY, USA, 394–405. doi:10.1145/3756681.3756964

[3] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2015. Are test smells really harmful? an empirical study. *Empirical Software Engineering* 20, 4 (2015), 1052–1094.

[4] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David W. Binkley. 2012. An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Maintenance. In *28th IEEE International Conference on Software Maintenance (ICSM)*.

[5] Keila Lucas, Rohit Gheyi, Elvys Soares, Márcio Ribeiro, and Ivan Machado. 2024. Evaluating Large Language Models in Detecting Test Smells. In *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 672–678. doi:10.5753/sbes.2024.3642

[6] Yuetian Mao, Junjie He, and Chunyang Chen. 2025. From Prompts to Templates: A Systematic Prompt Template Analysis for Real-world LLMapps. In *Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE)*. ACM. Industry Track.

[7] Simon Ott, Adriano Barbosa-Silva, Kathrin Blagec, Jan Brauner, and Matthias Samwald. 2022. Mapping global dynamics of benchmark creation and saturation in artificial intelligence. *Nature Communications* 13, 1 (2022), 6793.

[8] Anthony Peruma, Khalid Saeed Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2019. On the distribution of test smells in open source android applications: An exploratory study. (2019).

[9] Mark Richards and Neal Ford. 2020. *Fundamentals of software architecture: an engineering approach*. O'Reilly Media.

[10] Railana Santana, Luana Martins, Márcio Ribeiro, and Ivan Machado. 2025. An Empirical Study on the Co-occurrence of Test Smells. In *Anais do X Simpósio Brasileiro de Testes de Software Sistemático e Automatizado* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 104–113. doi:10.5753/sast.2025.14387

[11] Shreya Shankar, JD Zamfirescu-Pereira, Björn Hartmann, Aditya Parameswaran, and Ian Arawjo. 2024. Who validates the validators? aligning llm-assisted evaluation of llm outputs with human preferences. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. 1–14.

[12] Annalisa Szymanski, Simret Araya Gebreegziabher, Oghenemaro Anuyah, Ronald A. Metoyer, and Toby Jia-Jun Li. 2026. Designing Staged Evaluation Workflows for LLMs: Integrating Domain Experts, Lay Users, and Model-Generated Evaluation Criteria. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems (CHI '26)*. Association for Computing Machinery, New York, NY, USA, Article 616, 20 pages. doi:10.1145/3772318.3790897

[13] Arie van Deursen, Leon M. F. Moonen, Alex van den Bergh, and Gerard Kok. 2001. Refactoring Test Code. *CWI (Centre for Mathematics and Computer Science)* (2001).

[14] Tássio Virgínio, Railana Santana, Luana Almeida Martins, Larissa Rocha Soares, Heitor Costa, and Ivan Machado. 2019. On the Influence of Test Smells on Test Coverage. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering (SBES '19)*. ACM, 467–471. doi:10.1145/3350768.3350775

[15] Tássio Virgínio, Luana Martins, Larissa Rocha, Railana Santana, Adriana Cruz, Heitor Costa, and Ivan Machado. 2020. JNose: Java Test Smell Detector. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering* (Natal, Brazil) *(SBES '20)*. Association for Computing Machinery, New York, NY, USA, 564–569. doi:10.1145/3422392.3422499

[16] Tássio Virgínio, Márcio Ribeiro, and Ivan Machado. 2025. On the prevalence of test smells in mobile development. In *Anais do X Simpósio Brasileiro de Testes de Software Sistemático e Automatizado* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 84–93. doi:10.5753/sast.2025.14327

[17] Mo Wang, Minjuan Wang, Xin Xu, Lanqing Yang, Dunbo Cai, and Minghao Yin. 2023. Unleashing ChatGPT's power: A case study on optimizing information retrieval in flipped classrooms via prompt engineering. *IEEE Transactions on Learning Technologies* 17 (2023), 629–641.

[18] Ian Webster. 2023. promptfoo: Test your prompts. *https://www. promptfoo. dev/* (2023).

[19] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. 2024. *Experimentation in software engineering* (2 ed.). Springer. doi:10.1007/978-3-662-69306-3

[20] Di Wu, Fangwen Mu, Lin Shi, Zhaoqiang Guo, Kui Liu, Weiguang Zhuang, Yuqi Zhong, and Li Zhang. 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* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1345–1357. doi:10.1145/3691620.3695508