NoseSense: Benchmarking LLMs for Test Smell Detection

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

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

Letícia Ribeiro
Institute of Computing
Federal University of Bahia
Salvador, Brazil
leticiabarreto@ufba.br

Tássio Virgínio
Institute of Computing
Federal University of Bahia
Salvador, Brazil
tassio.virginio@ufba.br

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

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

Abstract

Unit test quality is a critical concern in software engineering, as anti-patterns in test code (known as test smells) compromise maintainability and reliability of test suites. The rapid evolution of Large Language Models (LLMs) has motivated their application to this domain, yet the pace of model releases renders point-in-time empirical studies quickly outdated. In this paper, we develop NoseSense, a benchmark designed to enable systematic and reproducible evaluation of LLMs for test smell detection. We mined 1,275 open-source Java repositories, applied the JNose static analysis tool to curate a balanced dataset of 1,100 real test smell instances, and evaluated six state-of-the-art models through a dedicated automated tool. The results indicate that model scale and generation significantly impact performance: Gemma-4-31B-it achieved the highest accuracy (67.36%), whereas the smaller-scale Gemma-3n-E4B-it achieved 38.55%. NoseSense provides a robust and reproducible foundation for monitoring LLM capabilities in software quality assurance, with a publicly available dataset, replication package, and benchmark implementation.

Keywords

LLM, Benchmark, Test Smells, Software Quality

1 Introduction

The rapid evolution of Large Language Models (LLMs), driven by the emergence of Transformer-based architectures such as BERT [5], PaLM [4], and LLaMA [19], has promoted significant advances in research focused on their application in various fields, such as medicine [9, 11], engineering [6, 8, 12], and computing [14, 23]. This is due to their ability to understand, reason, and communicate in natural language, and to solve diverse tasks in contrast to previous models that were limited to solving specific tasks [3, 15].

Research on the use of LLMs in Software Engineering has grown rapidly in recent years [15]. Among these studies, several focus on software quality, highlighting the potential of LLMs to support quality assurance activities. Among these, recent studies have explored the use of LLMs for test refactoring, test generation, and the detection of test smells and errors [17, 18, 22]. However, the rapid evolution of these models means that empirical findings can quickly become outdated as new and more capable versions are continuously released [3, 15]. Most existing studies rely on empirical comparisons between two or more LLMs to evaluate their performance on specific software engineering tasks [3, 22].

To evaluate LLMs, the literature commonly structures the evaluation process around four key elements: tasks, datasets, metrics, and benchmarks [3]. Among these, benchmarks play a central role by providing standardized environments for assessing and comparing model performance. More broadly, benchmarks are widely adopted to compare software and hardware systems, as well as to evaluate methodological approaches in a fair and reproducible manner [20].

To address this challenge, we develop a benchmark for evaluating and comparing multiple LLMs with respect to unit test quality. Specifically, the benchmark assesses the ability of LLMs to identify anti-patterns in unit test code, commonly referred to as *test smells*. These anti-patterns indicate deficiencies in test code that may compromise its maintainability, readability, or effectiveness over time [1].

To demonstrate the applicability of the proposed benchmark, we conducted a case study involving the most widely used LLMs available at the time of this study. The results reveal substantial performance differences among the evaluated models. Gemma-4-31B-it achieved the highest accuracy (67.36%), followed by GPT-OSS-120B (62.09%) and GPT-OSS-20B (58.73%), whereas Gemma-3n-E4B-it achieved the lowest accuracy (38.55%). To support reproducibility and enable the evaluation of future LLM releases, we publicly provide the dataset, replication package[1], and benchmark implementation[2].

To guide our empirical evaluation, we define three research questions. RQ1 investigates how effectively state-of-the-art LLMs identify and classify test smells. RQ2 compares the performance of different LLMs, both overall and for individual test smell types. Finally, RQ3 examines how model characteristics, including scale and architecture, influence performance in test smell detection.

The remainder of this paper is organized as follows. Section 2 presents the theoretical background on LLMs, benchmarking, and test smells. Section 3 reviews the related work. Section 4 describes the research methodology, covering data collection, static analysis, and prompt engineering. Section 5 presents the NoseSense tool and its evaluation pipeline. Section 6 presents and discusses the results.

[1] https://zenodo.org/records/20941814
[2] https://github.com/arieslab/NoseSense

Section 7 discusses the threats to validity, and Section 8 concludes the paper.

2 Background

2.1 Large Language Models

Large Language Models (LLMs) are language models trained using machine learning on large text corpora to generate appropriate responses in natural language based on a user's prompt. LLMs represent a paradigm shift in artificial intelligence, going beyond natural language processing (NLP) to understand, reason, and converse with humans about topics in various domains [2, 15].

The rapid evolution of LLMs has had a significant social impact and sparked great interest in both academia and industry, driven by the accessibility of tools like OpenAI's ChatGPT and Google's Gemini [25]. Because of their ability to solve a wide range of problems, the use and popularity of these technologies continue to expand, assisting research, increasing productivity in text-intensive tasks, and boosting the economy [3, 25]. At the same time, this widespread adoption increases the need for rigorous evaluation, as incorporating LLMs into critical workflows requires a clear understanding of their limitations, biases, and reliability.

In this work, the six language models evaluated were selected based on six main criteria to ensure a robust and comprehensive evaluation. First, we considered provider diversity by choosing three distinct families (GPT-OSS, Gemma, and Qwen) from different organizations such as OpenAI, Google, and Alibaba respectively, which reduces the biases inherent in a single training strategy or alignment method. Second, scale diversity was prioritized, spanning from a few billion parameters (e.g. Gemma-3n-E4B-it with ~4B parameters) to over 100 billion parameters (e.g. Qwen-3-235B and GPT-OSS-120B), allowing us to investigate whether increased capacity correlates with test smell detection capabilities. Third, within-family comparisons between models of the same family (such as GPT-OSS-120B vs. GPT-OSS-20B) to minimize training differences and isolate the scale variable. Fourth, architectural diversity by including both dense models and Mixture-of-Experts (MoE) approaches. Fifth, open and reproducible models with open weights or official availability to facilitate replication. Finally, relevance to code and reasoning of the models, as their specifications highlight advanced programming and structured reasoning capabilities, which are critical for analyzing complex Java code structures.

Table 1 provides the key characteristics and specifications of the evaluated language models.

Table 1: Key Characteristics and Specifications of the Evaluated Models

Model Name	Family	Provider	Architecture	Access	Params
GPT-OSS-120B	GPT-OSS	OpenAI	Dense	API (Open Weights)	120B
GPT-OSS-20B	GPT-OSS	OpenAI	Dense	API (Open Weights)	20B
Gemma-4-31B-it	Gemma	Google	Dense	API (Open Weights)	31B
Gemma-3n-E4B-it	Gemma	Google	Dense	API (Open Weights)	~4B
Qwen-3-235B	Qwen	Alibaba	MoE	API (Open Weights)	235B (22B active)
Qwen-3.5-9B	Qwen	Alibaba	Dense	API (Open Weights)	9B

2.2 Benchmarking

Benchmarks are tools designed to compare systems and components according to specific factors such as performance, effectiveness, or efficiency. In computer science, benchmarking has become well-accepted for evaluating and comparing competing products [20].

The main characteristics of a good benchmark include relevance, reproducibility, fairness, verifiability, and usability [20]. A robust benchmark guides the model development process while supports identify limitations, biases, and possible regressions that may arise during adaptation [15].

In the context of LLMs, benchmarking is essential because it allows us to quantify an LLM's performance across various tasks and domains using standardized metrics [7]. As LLMs are implemented in different spheres, benchmarking has evolved from static measurements of precision to multi-dimensional general-purpose evaluations [15].

2.3 Test Smells

Test smells are indicators of potential problems in test code, primarily found in unit tests [1, 26]. Although common, test smells represent a significant vulnerability for source code maintainability, as they hinder the programmer's understanding during maintenance activities [1]. Even though they are similar to code smells, Van Deursen et al. [21] point out that test smells have their own set of problems and solutions, classifying eleven distinct types. Other works have proposed new types of test smells [13] and verified their co-occurrence [16]. For the automatic detection of these patterns in Java code, we use JNose [24], a static analysis tool developed specifically to identify test smells in Java projects. JNose is used in Phase 2 of our methodology to generate the benchmark oracle (i.e., the set of correct labels for each code snippet evaluated by the LLMs) and is not part of the NoseSense execution platform itself. The evaluated test smell types are summarized in Table 2.

Table 2: Descriptions of Test Smell Types Evaluated

Test Smell	Description
Assertion Roulette	Multiple assertions without description, making it difficult to know which one failed.
Magic Number Test	Undocumented numerical parameters passed directly to assertions or methods.
Eager Test	Verifies multiple methods of the object under test, reducing the focus of the test.
Ignored Test	Inactive or disabled test in the suite via annotation (e.g., @Ignore).
Unknown Test	Test without any assertion, making its actual verification goal obscure.
Verbose Test	Excessively long test method (30 or more lines of code), increasing complexity.
Conditional Test Logic	Conditional logic (e.g., if/ else or loops), reducing determinism.
Duplicate Assert	Repeated verifications of the same condition redundantly within the same test.
Exception Catching	Test that relies on manual catching or throwing of exceptions.
Sensitive Equality	Fragile assertions based on textual representation (toString()).
None	Test free of smells, serving as control for the benchmark.

Motivated by the rapid evolution of LLMs and the limited lifespan of conventional empirical evaluations, this paper proposes a

benchmark for assessing the ability of LLMs to automatically detect test smells. Our approach builds on the recent advances of LLMs in code analysis and semantic understanding, exploring their potential to support test quality assessment. The benchmark enables a systematic evaluation of models from different providers and scales, making it possible to compare their effectiveness on the test smell detection task while also investigating the influence of prompt design. By providing a publicly available dataset, a reproducible evaluation methodology, and an extensible evaluation framework, the benchmark supports continuous comparisons as new models become available, contributing to research on automated test quality assessment.

3 Related Work

This section presents related work on the use of LLMs in software engineering, emphasizing research on software quality and benchmarking. By the time we wrote this work, we found no other benchmarks focused solely on evaluating LLMs for test smells. Therefore, we discuss studies that explore the application of LLMs in test code verification tasks, and works that propose benchmarks for evaluating models in this context.

The work of Velasco et al. [22] introduces CodeSmellEval, a benchmark aimed at measuring the propensity of LLMs to generate code smells in code generation tasks. For this purpose, the authors propose the Propensity Smelly Score (PSC) metric and provide a specialized dataset (CodeSmellData), structured to analyze model behavior under different types of smells. The study demonstrates the application of the benchmark on two reference LLMs, CodeLlama and Mistral, and discusses how the PSC goes beyond traditional accuracy metrics, offering a more direct view of the quality of code produced. These elements make their work relevant as a methodological basis for evaluations addressing the quality and reliability of code generated by LLMs [22].

With a focus on anti-pattern detection, the work of Silva et al. [18] presents an initial exploration of the effectiveness of ChatGPT in identifying code smells in Java projects. The authors utilize a dataset covering four types of smells (Blob, Data Class, Feature Envy, and Long Method), alongside prompts with different levels of specificity, and evaluate performance using precision, recall, and F-measure metrics. The study shows that the instruction format influences results and indicates higher effectiveness in detecting smells of critical severity, which reinforces the potential (and limitations) of using LLMs in code quality tasks [18].

Another test-focused benchmark was developed by Silva et al. [17], comparing large-scale LLMs, such as ChatGPT and Gemini, with small-scale open-weights models in terms of the number of generated tests and their quality. Their results show that some small-scale models are comparable to large-scale ones, producing a similar quantity of tests, with fewer smells and better mutation scores in certain cases [17].

Our work differs from previous studies in three main aspects. First, while Velasco et al. [22] evaluate the propensity of LLMs to generate code smells and Silva et al. [17] focus on the quantity and quality of tests generated by LLMs, our benchmark addresses the inverse problem: the capacity of the models to detect pre-existing test smells in the source code of real projects. Second, unlike Silva et al. [18], who analyze general code smells in Java using a single model (ChatGPT), we expand the scope to multiple state-of-the-art LLMs and restrict the analysis to the category of test smells, providing a more focused and reproducible evaluation framework. Third, we anchor the benchmark dataset on real open-source projects, ensuring ecological validity, and we encapsulate the entire evaluation pipeline, from sending prompts to collecting results, in an automated and reusable benchmarking application. This design simplifies the integration of new models, allowing the benchmark to keep pace with the rapid evolution of LLMs.

4 Methodology

In this section, we present the research methodology, covering data collection, processing, curation, and prompt engineering. The objective is to provide a clear and replicable view of the experimental design, ensuring transparency and robustness of the benchmark.

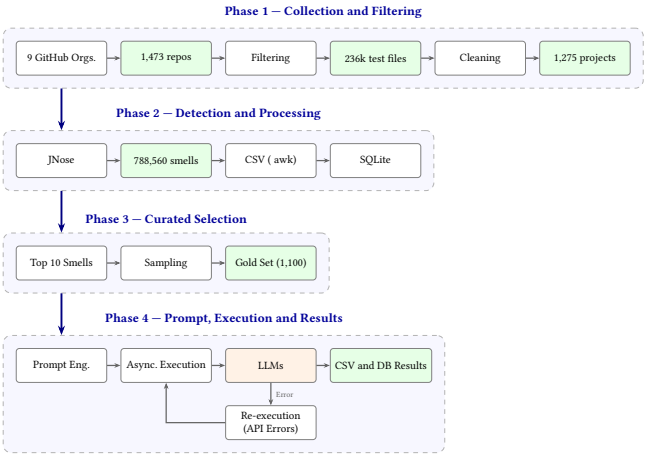

Figure 1: Overview of the NoseSense study methodology

4.1 Collection and Filtering

4.1.1 Selection and Cloning. The experiment comprises four main phases: (1) Collection and Filtering, (2) Detection and Processing, (3) Dataset Curation, and (4) Execution and Analysis. Each phase is composed of specific steps, which will be detailed below.

As Figure 1 shows, the initial phase of this study consisted of selecting the repositories that make up the dataset. To select the projects, we first randomly selected organizations that had repositories in the Java language on Github, in total, there were 9 organizations, which are listed in Table 3. We focused on Java projects because Java is the programming language most frequently studied in the test smell literature [1, 13]. Moreover, Java benefits from mature static analysis tools, such as JNose [24], which facilitate the construction and validation of the benchmark.

Data extraction and processing were carried out using the Python Data Analysis Library (Pandas for short)[3], allowing structured reading of the listed URLs. Interaction with the version control system

[3]https://pandas.pydata.org/

Table 3: Organizations from which repositories were collected

Organização	URL (GitHub)
Apache	https://github.com/apache
Elastic	https://github.com/elastic
Eclipse	https://github.com/eclipse
Google	https://github.com/google
Hibernate	https://github.com/hibernate
JUnit	https://github.com/junit-team
Mockito	https://github.com/mockito
Spring	https://github.com/spring-projects
Square	https://github.com/square

was implemented via the native subprocess module, which iteratively executed the `git clone` command for each valid entry. The script included exception handling mechanisms to manage inaccessible repositories or invalid links, ensuring that occasional failures did not interrupt the execution flow.

At the end of the procedure, the successful projects were stored locally in the "Projects" directory, constituting the raw base for the subsequent filtering and analysis steps.

4.1.2 Filtering. After the cloning step, the tracking and cataloging of test files contained in the repositories was carried out. The extraction was performed using the find utility (POSIX standard[4]) for a recursive scan in the directory tree of each project. The procedure was automated as shown in the code:

```
find . -type f -name "*Test*.java" > file.csv
find . -type f -name "*test*.java" >> file.csv
```

Listing 1: Bash command to list Java test files and save to CSV.

The script's operation followed strict criteria to ensure the integrity of the collection. The search was started from the root directory (.), using the `-type f` predicate to restrict the results exclusively to regular files, eliminating noise such as directories or symbolic links that could coincide with the naming.

For file identification, the `-name` parameter was used with matching patterns (globbing) protected by quotes, ensuring that the interpretation of wildcards was performed by the utility during recursion, and not by the shell. The data persistence strategy used output stream redirection in two steps: first, the overwrite operator "`>`" initialized the file `file.csv` with the results of the PascalCase convention (Test); second, the append operator "`>>`" added the results of the lowercase convention (test) to the end of the same file without overwriting the previously collected data.

This incremental approach allowed all naming variations to be consolidated into a single dataset, resulting in the final identification of 236,740 test files.

4.1.3 Directory Cleaning. Still in Phase 1, as shown in Figure 1, with the aim of ensuring the relevance of the sample and optimizing storage resources, the cloned repositories that did not contain verifiable test artifacts were removed. To operationalize this step, a Bash automation script was developed, designed to compute the difference between the set of physically present directories and the subset of previously validated projects.

The cleaning algorithm was structured in four main logical steps. First, qualified project identification processed the `list-tests.csv` file to extract unique names (in the organization/project format) of repositories that have been proven to contain test files. Second, local mapping scanned the file system to list all currently cloned project directories. Third, discrepancy calculation applied set operations using the grep tool to identify the symmetric difference, isolating repositories that existed locally but were not on the list of qualified projects. Finally, controlled deletion physically removed directories identified as invalid, incorporating a mandatory manual validation step as a safety measure before executing the permanent deletion command.

The application of this procedure resulted in a significant reduction of the sample space, eliminating noise in the database. From the initial total of 1,473 cloned repositories, 198 were discarded due to the absence of test coverage, resulting in a final consolidated dataset of 1,275 projects.

4.2 Detection and Data Processing

4.2.1 Abstract Syntax Tree (AST) Analysis Tool. For the step of identifying and classifying test smells, the JNose tool [24] was used. JNose is a static analysis solution designed to detect the presence of anti-patterns in Java test code, available in an open repository[5]. The tool performs parsing of the source code and generates structured reports, detailing the location and typology of the anomalies found.

Considering the volume of the dataset of 1,275 repositories, a batch processing strategy was adopted. The projects were segmented into groups of 100 directories per execution (with the residual batch containing 75 projects), aiming to optimize memory resources and mitigate runtime failures. This procedure resulted in the generation of 13 output files in CSV format, containing detailed metadata for each occurrence.

Each record generated by JNose comprises an attribute schema that ensures data traceability, containing: context and location metadata such as the project name (projectName), test file identification (name), absolute path (pathFile), and the production class under test (productionFile); software metrics including the JUnit framework version (junitVersion), lines of code (loc), and method count complexity (qtdMethods); test smell data consisting of taxonomic classification (testsmellname), affected method (testSmellMethod), and exact line range (testSmellLinebegin, testsmellLineend); and integrity signatures such as the analyzed code snippets (methodCode) and verification hashes (methodNameHash, methodcodeHash, fullHash) to ensure the uniqueness and integrity of the samples.

4.2.2 Consolidation and Structuring. The subsequent step consisted of unifying the data fragments and modeling them for a relational database, facilitating complex analytical queries.

The 13 partial files were concatenated into a single consolidated volume (`file_unify.csv`), totaling approximately 1.1 GB. To ensure the structural integrity of the final file, a script based on the

[4]IEEE Std 1003.1 – The Open Group Base Specifications: https://pubs.opengroup.org/onlinepubs/9699919799/

[5]Available at: https://github.com/arieslab/jnose

awk utility was used to remove redundant headers during concatenation, as detailed in Code 2.

```
awk 'FNR==1 && NR!=1{next;}{print}' *.csv > file_unify.csv
```

Listing 2: CSV file unification and sanitization script

The logic FNR==1 && NR!=1{next;} instructs the processor to ignore the first line (header) of all files subsequent to the first, preserving only the original column structure. After a manual sanitization step to remove any residual artifacts, the data were migrated to a SQLite database. The insertion was performed via CLI, using formatting directives for compatibility with the semicolon delimiter generated by the analysis tool:

```
sqlite3 metrics.db ".mode csv" ".separator ;" ".import file_unify.csv testsmells"
```

Listing 3: Data insertion procedure in SQLite

4.3 Final Sampling Curation

4.3.1 Distribution Analysis and Selection. The static analysis execution resulted in the identification of 788,560 instances of test smells distributed across the analyzed repositories. The frequency analysis revealed a predominance of issues related to assertions and undocumented constants, denominated the Magic Number test smell.

To define the scope of the benchmark, the criterion that was established to select the ten types of test smells was their incidence, selecting the ten that appeared the most during the use of JNose with the selected repositories shown in Table 3. However, a qualitative filter was applied to this selection: the Lazy Test pattern, although the fourth most frequent with 51,972 occurrences, was excluded from the sample.

The Lazy Test is characterized by a test method that verifies multiple functionalities simultaneously. Its correct detection requires a deep understanding of the call flow of production classes, introducing a high level of contextual noise. This complexity makes it difficult to isolate causality in LLM experiments. To maintain the cardinality of the set (Top 10) after this exclusion, the Sensitive Equality smell was included in the selection list due to it being the eleventh most frequent. Table 4 presents the final distribution.

It is worth noting that the tool detected nine other patterns that make up the long tail of the distribution, totaling 73,486 occurrences not included in the study, such as General Fixture, Resource Optimism, and Mystery Guest.

4.3.2 Extraction of Isolated Samples. The final curation step aimed to create a balanced Gold Set for the evaluation of LLMs. A uniform stratified sampling was defined, consisting of 100 code examples for each of the 10 selected types of test smells.

The extraction was automated via a Python script interacting with the SQLite database. A strict exclusivity criterion was applied: only test methods containing a single instance of a type of test smell (single-instance) were selected. This methodological restriction is crucial to eliminate ambiguity during inference, ensuring that the LLM's performance is evaluated on a specific defect, without the interference of multiple competing anti-patterns in the same context.

Table 4: Distribution of Test Smells Selected for the Benchmark

Rank	Test Smell Type	Occurrences
1	Assertion Roulette	254,494
2	Magic Number Test	163,642
3	Eager Test	56,555
4	Ignored Test	54,729
5	Unknown Test	53,257
6	Verbose Test	42,404
7	Conditional Logic Test	28,957
8	Duplicate Assert	27,086
9	Exception Catching/Throwing	16,975
10	Sensitive Equality	16,975
	Total Sampled	**715,074**

The resulting artifacts were exported as plain text files (.txt), constituting the final corpus for the execution of the experiments.

4.4 Prompt Engineering and Standardization

The validity of a benchmark involving LLMs depends intrinsically on the quality and consistency of the instructions provided to the models. To ensure the replicability of the experiments and align the construction of prompts with industry practices, this work was based on the systematic study conducted by Mao et al. [10].

4.4.1 Template Structure. Mao et al. [10] analyzed the construction of prompt templates in real applications based on LLMs, identifying that standardized structuring is critical to reduce operational instability. Based on the analysis of a vast dataset of open-source and industrial applications, the authors categorized the essential components that maximize the effectiveness of the models. Therefore, we adopted the CRISPE (Capacity and Role, Insight, Statement, Personality, and Experiment) framework derived from the taxonomy and statistical insights of this study for the composition of the NoseSense prompt, focusing on the components of greatest practical relevance: **Capacity and Role (C)**, which defines a "persona" that the AI must assume (present in 28.4% of industrial templates to contextualize model behavior and activate specific domain knowledge); **Insight (I)**, which provides context and background information to reduce ambiguity in task interpretation (identified in 56.2% of templates); **Statement (S)**, which represents the central directive to explicitly state the task's intention in an imperative manner (the most ubiquitous component, appearing in 86.7% of prompts); **Personality (P)**, which specifies the output format and style to ensure predictable and parser-compatible content (present in 39.7% of cases); and **Experiment (E)**, which includes final suggestions or recapitulations (considered optional and omitted here).

4.4.2 Prompt Implementation in NoseSense. Based on the empirical evidence from Mao et al. [10] regarding the effectiveness of these components in real scenarios, the framework was adapted to transform the task of detecting test smells into a deterministic classification. Below, the technical mapping between the theoretical components validated by the study and their implementation in the experiment is detailed.

For Capacity and Role (C), the persona was defined with the instruction " You are a senior software engineer...". Aligned with the practice of defining the agent's role [10], this component performs contextual preparation in the model's latent space, directing generation toward rigorous code quality criteria and mitigating naive interpretations.

For Insight (I), the context was established by including detailed Test Smell definitions followed by Java code examples. Given the high prevalence of this component in LLMapps (56.2%) [10], we use the In-Context Learning (ICL) technique to delimit the semantic scope of inference, ensuring that the model evaluates the code based on the study's taxonomic definitions and reducing hallucinations.

For the Statement (S), the execution directive is formulated as "From the definitions above, ANSWER the following question by choosing only ONE alternative...". Being the most critical component (86.7% frequency) [10], the directive uses imperative verbs and explicit constraints to convert the open task into a closed classification, forcing correlation between the input and the provided categories.

Finally, for Personality (P), format restriction was enforced via "Answer format: {A}". The imposition of a strict format limits the entropy of the response, ensuring precise automated extraction of responses in a large-scale benchmark, which is vital for interoperability as highlighted by Mao et al. [10]. Figure 2 illustrates the structured prompt template submitted to the LLMs.

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 other smell definitions with examples)

Persona (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: Magic Number Test
B: Duplicate Assert
C: Verbose Test
D: Ignored Test
E: None

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

Figure 2: Simplified example of a prompt submitted to LLMs. Options A–D are randomized per sample; option E is fixed.

5 The NoseSense Tool

To operationalize the evaluation pipeline, we developed NoseSense, a benchmarking tool implemented in Python using the LangChain library for model orchestration and the *asyncio* module for concurrent execution, mitigating the latency inherent in API calls.

5.1 Execution and Automation

The evaluation is executed in a pipeline of five main stages, as Figure 3 shows. The pipeline is designed to process the dataset efficiently while introducing controlled stochasticity in prompt construction to reduce positional bias and prevent pattern memorization by the evaluated models.

First, code ingestion and extraction recursively reads the artifact directory (test_smell_docs). For each text file, regular expressions (regex) are used to individually extract the Java code blocks delimited by markdown fences (" "), associating them with their ground truth (the original smell type).

Second, stochastic prompt engineering builds prompts dynamically to prevent models from memorizing patterns or exhibiting positional bias (a tendency to always choose the first or last option). To achieve this, the system performs distractor selection by randomly choosing three incorrect smell types from the available definitions, combining them with the unique correct answer and one "empty" option called "None" to form a set of five candidates in an option pool. It then shuffles the options randomly to break predictable patterns and maps the correct answer dynamically to one of the letters (A, B, C, D, or E), ensuring a uniform distribution across the answer key. This shuffling process is illustrated in Figure 4.

Third, asynchronous multi-model inference uses an asynchronous event loop (`asyncio.gather`) to submit a single test instance simultaneously to all evaluated models (Table 1), which underutilizes network latency compared to sequential executions and drastically reduces the total experiment time.

Fourth, response parsing and validation processes the textual responses from the LLMs using a validation layer based on regular expressions (`r'([A-E])'`). This captures the letter corresponding to the model's choice, ignoring noise or additional explanations that violate the formatting directive. Parsing failures or API errors are logged as handled exceptions to ensure that the batch flow is not interrupted.

Finally, result persistence incrementally saves the raw data, containing the response inferred by each model and the official answer key, in a structured SQLite database.

5.2 NoseSense Architecture and Implementation

To automate and standardize the execution of the evaluation experiments, we developed NoseSense, a benchmarking tool with a decoupled client-server architecture. NoseSense isolates the evaluation concerns and handles communication with remote LLM providers, result persistence, and real-time visualization of metrics. Figure 5 illustrates the high-level architecture of NoseSense.

NoseSense is organized into four main architectural layers. The Frontend (Client Layer) is developed using React and Next.js, implementing the App Router paradigm. It provides an analytical dashboard showing the real-time execution progress, overall and per-smell accuracies, and confusion matrices. The Routing Layer (API Gateway) is implemented in Python using the FastAPI framework running on top of the Uvicorn ASGI server. It isolates request domains and exposes RESTful endpoints for configuration, alongside a Server-Sent Events (SSE) stream for real-time experiment

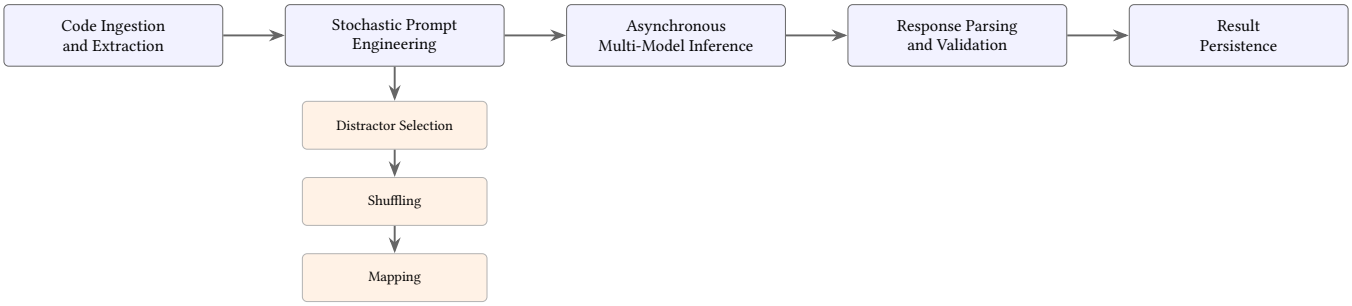

Figure 3: Execution and Automation flow: stages and sub-steps of the evaluation pipeline

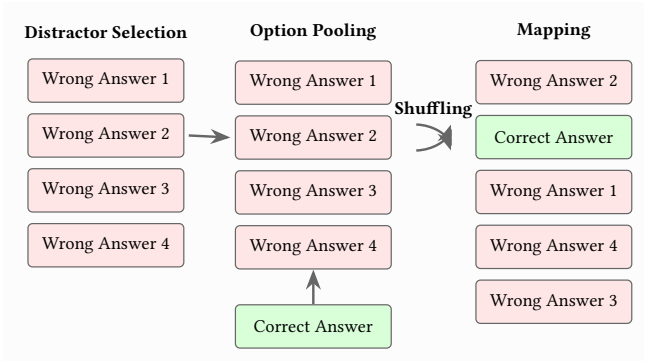

Figure 4: Stochastic Prompt Engineering Option Shuffling and Mapping Process

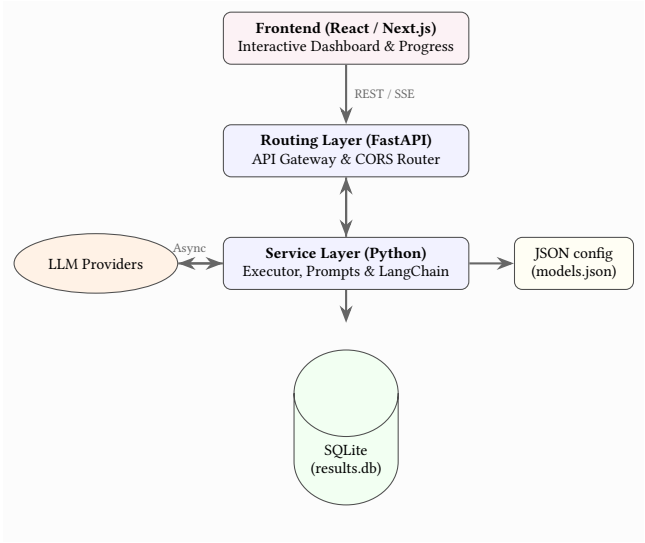

Figure 5: NoseSense Decoupled Client-Server Architecture

updates. The Service Layer (Business Logic) is the core engine of the tool and consists of three components: `llm_initializer.py` uses LangChain to configure LLM wrappers, `prompt_engine.py` formats prompts and performs option randomization, and `executor.py` manages concurrent asynchronous API calls using semaphores. Finally, the Persistence Layer employs a hybrid storage approach where a JSON file (`models.json`) stores provider configurations, and a local relational SQLite database (`results.db`) persists raw evaluations, option configurations, and metrics.

5.3 Operational Workflow

The operational workflow of NoseSense follows a sequence of five main phases. First, during initialization, the server is launched on port 8001, and local configurations and API keys are read from JSON files. Second, in the connection & handshake phase, the frontend initiates a connection and opens a Server-Sent Events (SSE) stream channel to monitor the progress. Third, during batch preparation, the backend loads the 1,100 Gold Set test cases from disk and registers a new session with a unique `run_id` in the database. Fourth, in the prompt-inference loop, the prompt is randomized for each test case and sent concurrently to the selected LLMs, while keep-alive packets are periodically sent to the frontend. Finally, during consolidation, model responses are parsed using regular expressions, matched against the ground truth, recorded in SQLite, and exported to a consolidated CSV report before closing the SSE stream.

The entire evaluation experiment was orchestrated from a local server running Ubuntu 24.04 LTS, equipped with an Intel Core i5-10500 processor (3.10GHz, 12 threads) and 16 GB of RAM, ensuring that execution bottlenecking was limited only by external API latency.

6 Results

In this section, we present the quantitative analysis of the performance of the evaluated LLMs in Test Smell detection. The raw data were processed to extract metrics of effectiveness (accuracy) and robustness (API failure rate and format compliance). To answer our research questions (RQs) in a structured manner, we mapped specific metrics to each of them: overall accuracy and accuracy rate by smell type are used to answer RQ1 (overall effectiveness) and RQ2 (model comparison by category); and the pair-wise comparison among the models is employed to answer RQ3 (relationship between scale, architecture, and performance).

6.1 General Performance Overview

Figure 6 summarizes the results obtained for each evaluated model. The models are sorted in descending order of accuracy, revealing a clear relationship between model scale and generation on performance. Each bar represents the 1,100 evaluated instances, split into correct answers (green) and errors (red), with overall accuracy shown on the right.

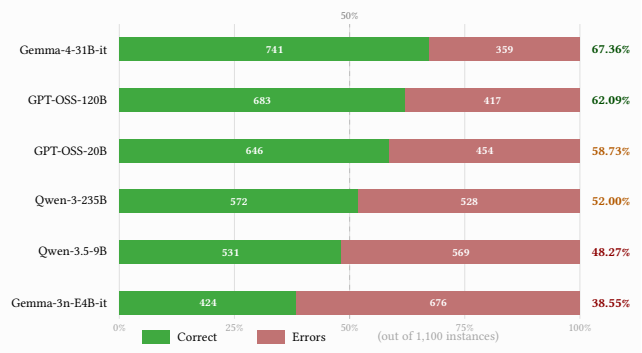

Figure 6: LLM Performance Overview: correct answers (green) and errors (red) out of 1,100 test instances. Counts shown inside bars; accuracy (%) on right. Dashed line marks 50%.

For a more granular understanding, Table 5 details the accuracy rate of each model for the ten evaluated test smell types and the clean test class (None). This breakdown allows us to identify which specific defects present greater complexity for the models compared to the overall average.

Table 5: Accuracy rate (in %) by type of Test Smell

Test Smell	GEM4-31	GPT120	GPT20	QWEN235	QWEN9	GEM3-E4
Assertion Roulette	92	88	67	84	20	21
Conditional Test Logic	88	57	56	60	45	80
Duplicate Assert	82	67	70	73	74	48
Eager Test	26	34	35	10	14	11
Exception Catching/Throwing	86	86	86	94	79	88
Ignored Test	16	10	8	11	13	20
Magic Number Test	69	67	80	56	41	25
Sensitive Equality	82	80	50	30	84	12
Unknown Test	33	40	39	48	38	13
Verbose Test	74	69	65	21	52	55
None	93	85	90	85	71	51

To further analyze classification errors, Table 6 presents the combined confusion matrix obtained by aggregating the predictions of all six evaluated models.

The evaluated models exhibited different levels of performance. Gemma-4-31B-it obtained the highest overall accuracy (67.36%). Its strongest results were observed for Assertion Roulette (92%) and Conditional Test Logic (88%), whereas Eager Test (26%) and Ignored Test (16%) proved to be the most challenging categories.

GPT-OSS-120B achieved the second-highest overall accuracy (62.09%), followed closely by its smaller counterpart, GPT-OSS-20B (58.73%). The modest performance difference between the two models (3.36 percentage points) suggests that increasing model size within the GPT-OSS family provides only limited gains for this specific task.

Table 6: Aggregated Confusion Matrix for the Evaluated LLMs

Actual / Predicted	AR	MN	ET	IT	UT	VT	CL	DA	EC	SE	N
AR	372	34	16	10	9	8	20	21	9	13	88
MN	41	338	12	4	10	7	12	18	12	0	146
ET	13	27	130	9	10	7	10	10	12	6	366
IT	22	38	27	78	32	38	17	16	29	12	291
UT	7	90	17	7	211	7	20	13	12	8	208
VT	35	22	24	6	25	336	22	11	16	4	99
CL	12	49	13	5	10	8	386	24	6	3	84
DA	58	13	11	1	9	8	12	414	16	2	56
EC	2	10	9	0	9	2	10	1	519	2	36
SE	2	14	8	2	13	0	6	5	1	338	211
N	9	21	15	2	22	8	9	6	23	10	475

Within the Qwen family, Qwen-3-235B, which adopts a MoE architecture, achieved an overall accuracy of 52.00%, outperforming the smaller dense model Qwen-3.5-9B (48.27%). Among all evaluated models, Gemma-3n-E4B-it obtained the lowest overall accuracy (38.55%). This result suggests that smaller models may face greater challenges in detecting test smells, although differences in architecture and model generation may also have contributed to the observed performance gap.

6.2 Discussion: Answering the Research Questions

RQ1: How effective are state-of-the-art LLMs in identifying and classifying test smells?
The evaluated models demonstrated moderate to high effectiveness depending on the model scale, with the best-performing model (Gemma-4-31B-it) achieving an accuracy of 67.36%. However, accuracy drops significantly in smaller models (reaching as low as 38.55%), indicating that test smell detection remains a complex semantic classification task.

RQ2: How do different LLMs compare in their overall performance and on specific types of test smells?
The comparison revealed that Gemma-4-31B-it and GPT-OSS-120B lead the classification. Across all models, smells with clear syntactic indicators, such as Exception Catching/Throwing, were consistently identified with high accuracy (up to 94% for Qwen-3-235B). In contrast, more subjective or context-dependent smells, such as Eager Test and Ignored Test, presented much lower accuracy rates across all models, indicating that these patterns are significantly more difficult for LLMs to detect.

RQ3: What is the relationship between model variables (such as scale and architecture) and their performance in test smell detection?
The results indicate that the relationship between parameter scale and performance is not linear and depends strongly on the model family, generation, and architecture. Within the GPT-OSS family, the 120B model outperformed the 20B model by only 3.36 percentage points (62.09% vs. 58.73%), despite a sixfold increase in parameter count. A similar trend was observed within the Qwen family, where Qwen-3-235B achieved an accuracy 3.73 percentage points higher than Qwen-3.5-9B (52.00% vs. 48.27%). However, comparing Gemma-4-31B-it (67.36%) to Gemma-3n-E4B-it (38.55%) revealed the largest performance gap observed in this study (28.81 percentage points), suggesting that improvements introduced across model

generations may have a greater impact on test smell detection than increases in model size alone.

To visually analyze this behavior, Figure 7 plots the parameter count of each model against its overall accuracy on a logarithmic scale, highlighting the performance trends within each model family.

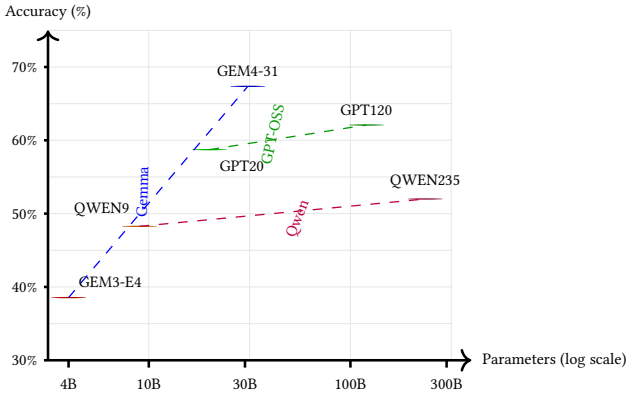

Figure 7: Model Parameters vs. Overall Accuracy (X-Axis Log Scale)

This visualization highlights that while increasing size within the same family yields diminishing returns, switching to a more modern model generation (e.g., Gemma-4-31B-it vs. Gemma-3n-E4B-it) leads to substantial improvements. This highlights the importance of training data quality and architectural refinements over raw scale.

Smells with clear syntactic markers were consistently detected across all models. Exception Catching/Throwing achieved the highest aggregate accuracy (86%), as try/catch blocks in test code provide a strong and locally recognizable structural signal. Similarly, Duplicate Assert (69%) and Assertion Roulette (62%) benefit from repetitive patterns that are straightforward to identify within a self-contained snippet, indicating that LLMs reliably exploit local syntactic features for these categories.

In contrast, smells requiring semantic reasoning beyond the immediate snippet proved substantially harder. Ignored Test achieved the lowest aggregate accuracy (13%); in Java, ignored tests manifest through multiple distinct patterns (such as different framework annotations, disabled assumptions, or conditional execution), and LLMs likely failed to recognize patterns beyond the standard cases. Eager Test (22%) is particularly challenging, as detecting it requires recognizing that a test exercises multiple methods of the class under test, information not always inferrable from the snippet alone. Unknown Test (35%) demands detecting the absence of assertions, a negative structural property with few notable lexical cues. Collectively, these patterns reveal a fundamental limitation of snippet-level evaluation: context-sensitive smells are systematically underdetected.

Analysis of the aggregated confusion matrix reveals a systematic conservative bias: misclassifications predominantly fall into the None class across most smell categories. This tendency is especially pronounced for Eager Test, where 366 of 600 aggregated instances were classified as None despite carrying a smell, and for Ignored Test, where 291 of 600 were similarly misclassified. In practice, this means LLMs tends to commit mistakes toward reporting smelly code as clean, producing more false negatives than false positives, a pattern with direct implications for their integration into automated quality assurance pipelines.

From a cost-efficiency standpoint, the modest performance gap between GPT-OSS-120B and GPT-OSS-20B (3.36 percentage points) suggests that practitioners can prefer smaller, less expensive models with minimal accuracy trade-off for this task. Compared to prior evaluations of LLMs on general code smells in Java [18], our multi-model benchmark reveals considerably higher performance variation across models, indicating that model selection is a critical factor specifically for test smell detection tasks.

7 Threats to Validity

Several threats were identified. Regarding construct validity, a single CRISPE template with multiple-choice options was used, which may limit generalizability and introduce response bias by constraining models to predefined alternatives; default inference hyperparameters (temperature, top-p) may also affect response determinism. Regarding external validity, results are limited to Java code and to a dataset of 1,100 instances drawn from nine large, mature organizations (e.g., Apache, Google, Spring), which may not represent smaller or academic projects, nor capture the full diversity of real-world test patterns. Regarding internal validity, dataset design choices (100 instances per category, exclusion of multi-smell instances, and omission of Lazy Test) were made for practical balance and may not fully reflect real smell distributions. Finally, JNose is used as the ground truth oracle; as a static analysis tool, it is subject to false positives and negatives, so the reported accuracy reflects agreement with JNose rather than absolute correctness, and any labeling errors propagate across all results. We plan to complement these findings with a user study involving LLM researchers to assess NoseSense as an evaluation framework.

8 Conclusion

This paper presented a benchmark for evaluating the capacity of LLMs to detect test smells in Java code. The methodology was structured into four main phases: repository collection and filtering, test smell detection and processing, sample curation, and automated experiment execution. Each phase was grounded in established software engineering and static analysis practices to ensure the reproducibility and transparency of the process.

Our results show that LLMs can identify test smells with varying degrees of accuracy, with the best model (Gemma-4-31B-it) achieving 67.36% overall accuracy. The benchmark proved to be robust and flexible, serving as a reliable framework for systematic evaluation of LLMs in software quality tasks. NoseSense can be used by researchers and practitioners alike, providing a dynamic benchmarking that serves to showcase which LLMs are the most accurate in detecting test smells in the test code. Due to its dynamism, we expect NoseSense to be useful independently of new LLMs releases. We hope that NoseSense will be used by the community to evaluate and guide the evolution of LLMs in automated software testing and quality assurance.

For future work, we plan to evaluate the tool with a user study involving software developers and AI researchers to assess NoseSense as an evaluation framework for benchmarking. Other directions to expand this work involve: evaluating the model performance using open-ended prompts without predefined alternatives; incorporating alternative prompt engineering strategies and few-shot learning to evaluate their influence on classification accuracy; and implementing multi-language support by expanding the benchmark to include other languages like Python and Kotlin.

Artifact Availability

The NoseSense tool, including its source code, dataset, and replication package, is publicly available at: https://zenodo.org/records/20941814. The repository for the tool is available at: https://github.com/arieslab/NoseSense.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001; Fundação de Amparo à Pesquisa do Estado da Bahia (FAPESB), grant PIE0002/2022; and CNPq, grants 315840/2023-4, 403361/2023-0 and 140587/2024-1.

References

[1] 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.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[3] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A Survey on Evaluation of Large Language Models. *ACM Trans. Intell. Syst. Technol.* 15, 3, Article 39 (March 2024), 45 pages. doi:10.1145/3641289

[4] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. PaLM: Scaling Language Modeling with Pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113. http://jmlr.org/papers/v24/22-1144.html

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. doi:10.18653/v1/N19-1423

[6] Lea Höltgen, Sven Zentgraf, Philipp Hagedorn, and Markus König. 2025. Utilizing large language models for semantic enrichment of infrastructure condition data: a comparative study of GPT and Llama models. *AI in Civil Engineering* 4, 1 (2025), 14.

[7] Xing Hu, Feifei Niu, Junkai Chen, Xin Zhou, Junwei Zhang, Junda He, Xin Xia, and David Lo. 2025. Assessing and Advancing Benchmarks for Evaluating Large Language Models in Software Engineering Tasks. arXiv:2505.08903 [cs.SE] https://arxiv.org/abs/2505.08903

[8] Dimitrios Kampelopoulos, Athina Tsanousa, Stefanos Vrochidis, and Ioannis Kompatsiaris. 2025. A review of LLMs and their applications in the architecture, engineering and construction industry. *Artificial Intelligence Review* 58, 8 (2025), 250.

[9] Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. 2019. BioBERT: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics* 36, 4 (Sept. 2019), 1234–1240. doi:10.1093/bioinformatics/btz682

[10] 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.

[11] Xiangbin Meng, Xiangyu Yan, Kuo Zhang, Da Liu, Xiaojuan Cui, Yaodong Yang, Muhan Zhang, Chunxia Cao, Jingjia Wang, Xuliang Wang, et al. 2024. The application of large language models in medicine: A scoping review. *Iscience* 27, 5 (2024).

[12] Sushant Kumar Pandey, Sivajeet Chand, Jennifer Horkoff, Miroslaw Staron, Miroslaw Ochodek, and Darko Durisic. 2025. Design pattern recognition: a study of large language models. *Empirical Software Engineering* 30, 3 (2025), 69.

[13] 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).

[14] Dhivyabharathi Ramasamy, Cristina Sarasua, and Abraham Bernstein. 2025. AI support for data scientists: An empirical study on workflow and alternative code recommendations. *Empirical Software Engineering* 30, 5 (2025), 133.

[15] Seyed Mahmoud Sajjadi Mohammadabadi, Burak Cem Kara, Can Eyupoglu, Can Uzay, Mehmet Serkan Tosun, and Oktay Karakuş. 2025. A survey of large language models: evolution, architectures, adaptation, benchmarking, applications, challenges, and societal implications. *Electronics* 14, 18 (2025).

[16] 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

[17] Esdras Silva, Roberta Coelho, and Lyrene Silva. 2025. LLMs as Test Generators: A Comparative Benchmarking Study. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 25–36. doi:10.5753/sbes.2025.9618

[18] Luciana Lourdes Silva, Janio Rosa da Silva, Joao Eduardo Montandon, Marcus Andrade, and Marco Tulio Valente. 2024. Detecting Code Smells using ChatGPT: Initial Insights. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Barcelona, Spain) *(ESEM '24)*. Association for Computing Machinery, New York, NY, USA, 400–406. doi:10.1145/3674805.3690742

[19] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL] https://arxiv.org/abs/2302.13971

[20] Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. 2015. How to Build a Benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (Austin, Texas, USA) *(ICPE '15)*. Association for Computing Machinery, New York, NY, USA, 333–336. doi:10.1145/2668930.2688819

[21] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. Citeseer, 92–95.

[22] Alejandro Velasco, Daniel Rodriguez-Cardenas, Luftar Rahman Alif, David N. Palacio, and Denys Poshyvanyk. 2025. How Propense Are Large Language Models at Producing Code Smells? A Benchmarking Study. In *2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 96–100. doi:10.1109/ICSE-NIER66352.2025.00025

[23] Ashwin Prasad Shivarpatna Venkatesh, Rose Sunil, Samkutty Sabu, Amir M Mir, Sofia Reis, and Eric Bodden. 2025. An empirical study of large language models for type and call graph analysis in Python and JavaScript. *Empirical Software Engineering* 30, 6 (2025), 167.

[24] 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

[25] Yu Xie and Sofia Avila. 2025. The social impact of generative LLM-based AI. *Chinese Journal of Sociology* 11, 1 (2025), 31–57. doi:10.1177/2057150X251315997

[26] Vahid Garousi Yusifoğlu, Yasaman Amannejad, and Aysu Betin Can. 2015. Software test-code engineering: A systematic mapping. *Information and Software Technology* 58 (2015), 123–147.