# Towards LLM-Based Detection of Elixir Code Smells: An Empirical Evaluation

Mateus José Dias
Department of Informatics, Federal University of Viçosa (UFV)
Viçosa, Brazil
mateus.j.dias@ufv.br

Lucas Vegi
Department of Informatics, Federal University of Viçosa (UFV)
Viçosa, Brazil
lucas.vegi@ufv.br

## Abstract

Large Language Models (LLMs) have shown promise for code smell detection, but their effectiveness for functional programming languages remains largely unexplored. This paper presents the first empirical evaluation of LLM-based Elixir code smell identification. We introduce ExSmell-Train, a retrieval corpus, ExSmell-Gold, a labeled benchmark, and ExSmellBench, an experimental framework for evaluating four prompting strategies across the 35 smells of the Elixir smell catalog. We compare one proprietary and three open-weight LLMs under description-based and Retrieval-Augmented Generation (RAG) prompting. The results show that RAG improves multi-label smell identification only when combined with smell descriptions, whereas retrieved examples alone are insufficient. Performance varies substantially across smells, categories, and models: design-related Elixir smells are identified more reliably than traditional smells, and no model dominates across all criteria. Gemini 3.1 Flash Lite achieves the highest recall, whereas Qwen2.5-Coder 32B provides the best precision and overall agreement. Our publicly available datasets and benchmark establish a foundation for future research on LLM-based smell detection for functional languages.

## Keywords

Code Smells, Elixir, Functional Programming, AI4SE

## 1 Introduction

*Code smells* are recurring design structures that indicate potential maintainability problems and often motivate refactoring activities [5]. Automatic smell detection has been extensively studied for object-oriented languages, resulting in numerous techniques based on software metrics, static analysis [11], and, more recently, Large Language Models (LLMs) [10]. However, these approaches have been developed primarily for object-oriented systems, while functional programming languages remain largely unexplored.

Among functional languages, Elixir has gained increasing industrial adoption due to its scalability, fault tolerance, and concurrency model built on the Erlang VM [7]. Recent work identified a catalog of 35 Elixir code smells, including both traditional and language-specific quality problems [12]. Although LLMs have shown promising results for many software engineering tasks, including smell detection, their effectiveness for functional languages has not yet been systematically investigated. Likewise, the impact of Retrieval-Augmented Generation (RAG) on this task remains unknown.

To address this gap, we present the first empirical evaluation of LLM-based Elixir code smell identification. We introduce ExSmell-Train [3], a retrieval corpus, ExSmell-Gold [2], a labeled benchmark containing 3,500 Elixir source files, and ExSmellBench [4], an experimental framework for systematically evaluating LLM-based smell detection. Using these artifacts, we compare four prompting strategies, including RAG, on one proprietary and three open-weight code LLMs across the 35 smells of the Elixir smell catalog. The evaluation provides the first systematic evidence on the effectiveness of LLMs for Elixir smell detection, the impact of RAG, and the trade-offs between proprietary and open-weight models.

The contributions of this paper are fourfold: (i) the first empirical evaluation of LLM-based Elixir code smell identification; (ii) ExSmell-Train, ExSmell-Gold, and ExSmellBench, publicly available artifacts supporting reproducible research; (iii) a comprehensive comparison of four prompting strategies and four LLMs across the 35 smells of the Elixir smell catalog; and (iv) empirical evidence on the effectiveness and limitations of RAG for code smell detection in a functional programming language.

The remainder of this paper is organized as follows: background and related work (Section 2), methodology (Section 3), results (Section 4), practical implications (Section 5), threats to validity (Section 6), and conclusions and future work (Section 7).

## 2 Background and Related Work

This study adopts the Elixir code smell catalog proposed by Vegi et al. [12] as the reference taxonomy. The catalog defines 35 smell types, covering both traditional design problems and Elixir-specific smells related to OTP, processes, GenServers, Agents, macros, configuration, and pattern matching. It provides the label set used throughout this study for dataset construction and evaluation.

Recent work has shown that LLMs can support the construction of labeled software engineering datasets. For example, Kassem et al. [8] generate labeled semantic errors by prompting GPT-4.1 to inject bugs into correct Python programs. Similar to their work, we use LLMs to address the lack of labeled data. However, our focus is on Elixir code smells, whose labels require manual validation because, unlike semantic errors, they cannot generally be verified through program execution.

## 3 Methodology

This section describes our methodology. As no labeled benchmark for Elixir code smells was available, we created two independent datasets: one for retrieval and another for evaluation. Figure 1 summarizes the workflow, from dataset construction to retrieval, prompting, model execution, and performance analysis.

Our study addresses the following research questions:

**RQ1.** Does the use of RAG with semantically similar examples improve code smell detection compared with zero-shot prompts based only on smell descriptions?

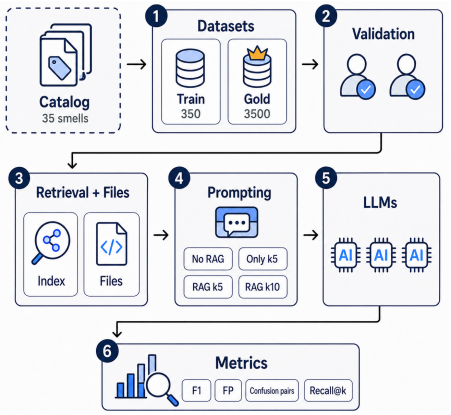


**Figure 1: Overview of the empirical methodology.**

**RQ2.** What is the impact of RAG on the frequency of incorrectly identifying smells in smell-free code?
**RQ3.** Which Elixir code smells are more easily detected by LLMs, and which remain more difficult?
**RQ4.** Does LLM performance vary across smell categories, such as design-related, low-level concerns, and traditional smells?
**RQ5.** Do LLMs exhibit consistent confusion patterns between specific smells in the Elixir smell catalog?
**RQ6.** Do larger or newer models provide consistent gains in Elixir code smell detection?

***1) Datasets creation:*** We constructed two datasets. The first, named ExSmell-Train [3], was used exclusively as the retrieval corpus for RAG-based prompting. It contains 10 validated examples for each of the 35 smells in the Elixir smell catalog, totaling 350 positive examples. The examples were generated with ChatGPT-5.4 using the smell catalog [12] as input, and were manually validated before being included in the retrieval corpus.

ExSmell-Gold [2] was used as the evaluation benchmark. To reduce generation overlap between retrieval and evaluation data, ExSmell-Gold was generated with a different LLM, Claude Sonnet 4.6. It contains 50 examples for each smell, totaling 1,750 positive examples, plus 1,750 smell-free examples, for a total of 3,500 Elixir files. Thus, the dataset is balanced at the binary level, although our main evaluation treats it as a multi-label smell identification task.

For each positive example, we stored a clean version containing only the Elixir code and an annotated version containing the expected smell location and explanation. Only the clean version was given to the evaluated models. Although each positive file was generated for a primary smell, files may contain additional smells; therefore, expected labels are represented as sets of smell names rather than as a single label.

***2) Dataset validation:*** Both datasets were independently inspected by the two authors, with disagreements resolved by discussion. For ExSmell-Train, each example was manually validated, and examples that lacked the expected smell were discarded and replaced until 10 validated examples per smell were obtained.

For ExSmell-Gold, we adopted a stratified sampling strategy to estimate label quality while keeping the validation effort feasible. This strategy was inspired by Kassem et al. [8], who validate LLM-generated labeled code datasets through a combination of automated checks and manual analysis of statistically sampled examples. Unlike semantic-error datasets that can often be partially validated through test execution, code smell labels require manual judgment because smells are maintainability symptoms rather than executable failures. Following a 90% confidence level, a 5% margin of error, and a conservative expected proportion of $p = 0.5$, we validated 43 examples for each of the 35 smell strata. We also validated 235 examples from the 1,750 smell-free files, treating them as a single negative population.

***3) Experimental pipeline:*** We implemented an experimental system, called ExSmellBench [4], in Elixir/Phoenix. The system loads files from ExSmell-Gold, constructs prompts according to the selected strategy, optionally retrieves examples from ExSmell-Train, sends requests to the selected LLM, stores the responses, and compares predicted labels with the expected labels.

For each analyzed file, ExSmellBench records the model, prompting strategy, execution status, duration, expected labels, predicted labels, predicted location, and explanation. Model outputs were required to follow a structured JSON format containing: whether the file has a smell, the predicted smell names, the detected location, and a brief explanation.

***4) Prompting strategies:*** We evaluated four prompting strategies: **`[No RAG]`** provides the target code and a summarized table of the 35 smells, but no retrieved examples. This is the description-only baseline; **`[Only-RAG k=5]`** provides the target code and five semantically similar examples retrieved from ExSmell-Train, but not the complete smell summary; **`[RAG k=5]`** provides the target code, the smell summary, and the five most similar examples from ExSmell-Train; and **`[RAG k=10]`** is equivalent to **`RAG k=5`**, but uses ten retrieved examples instead of five.

For all RAG-based strategies, we indexed ExSmell-Train using Gemini Embedding 1 and retrieved examples using cosine similarity. To increase diversity and avoid overrepresenting a single smell type, retrieval was constrained to include at most one example per smell. Retrieved examples always came from ExSmell-Train, while all evaluated files came from ExSmell-Gold.

***5) Models:*** We evaluated four LLMs: Gemini 3.1 Flash Lite, Qwen2.5-Coder 14B, Qwen2.5-Coder 32B, and Qwen3-Coder 30B. The Qwen models were executed locally through Ollama with an 8,192-token context window on a machine equipped with an Intel Core i7-13700 CPU, 32 GB of RAM, and an NVIDIA RTX 4060 GPU with 16 GB of VRAM. Gemini 3.1 Flash Lite was accessed through an API using its default context configuration. For all models, the remaining inference parameters were kept at their default values to avoid model-specific tuning effects.

***6) Evaluation Metrics:*** We evaluate predictions as a closed-set multi-label classification task over the 35 smells in the catalog, where smell-free files have an empty label set. Our main metrics are Macro Precision, Macro Recall, and Macro F1, since they give equal weight to each smell type. We also use exact match, Mean Jaccard, clean-code false positives, per-smell and category-level metrics, and confusion pairs as complementary analyses. For RAG strategies, we report Recall@$k$ to measure whether the expected smell appears among the retrieved examples.

## 4 Results and Discussion

This section presents the results by research question. We focus on multi-label smell identification and use complementary analyses to explain differences across strategies, smells, categories, and models.

**RQ1 — Does RAG with semantically similar examples improve smell identification over description-only prompting?**

> Yes. RAG improves smell identification across all models, but only when combined with smell descriptions.

Figure 2 summarizes Macro F1 for all model–strategy configurations. Compared with the description-only baseline, RAG consistently improves performance. With five retrieved examples, the mean Macro F1 increases by 0.127 across the four models; with ten examples, the mean gain is 0.130. Thus, RAG provides a substantial improvement over **`No RAG`**, but increasing from five to ten examples adds only a marginal gain of 0.003 Macro F1 on average.

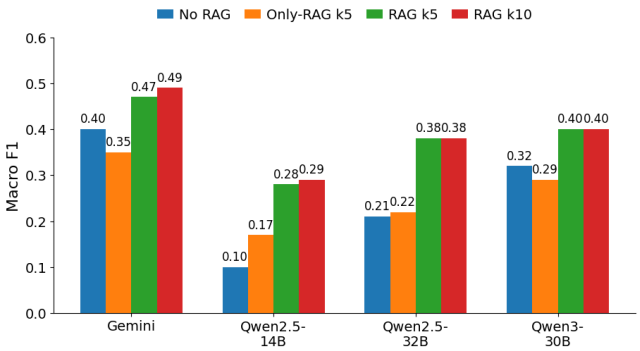


**Figure 2: Macro F1 for each model–strategy configuration.**

The largest improvement is observed for Qwen2.5-Coder 14B, whose Macro F1 increases from 0.103 (**`No RAG`**) to 0.291 (**`RAG k=10`**). The best overall result is achieved by Gemini 3.1 Flash Lite with **`RAG k=10`**, reaching 0.487 Macro F1. These results indicate that retrieved examples effectively improve smell detection, especially when the baseline model is weaker.

However, retrieved examples alone are not sufficient. **`Only-RAG k=5`** performs substantially worse than **`RAG k=5`**, with an average Macro F1 gap of 0.126. For Gemini, **`Only-RAG`** is even worse than **`No RAG`**, decreasing from 0.397 to 0.353 Macro F1. This suggests that the smell descriptions and the retrieved examples play complementary roles: the descriptions define the smell label space, while the examples help instantiate those concepts in code. Without the descriptions, the model becomes highly dependent on which smells appear among the retrieved examples.

The limited difference between **`RAG k=5`** and **`RAG k=10`** suggests that more retrieved examples are not necessarily better. Additional examples may increase the chance of retrieving the correct smell, but they may also introduce competing evidence, consistent with prior work showing that LLMs do not always use longer context effectively [9]. It also aligns with LLM-based code refinement studies in which more explicit task guidance, rather than simply more prompt content, improves model behavior [6].

**RQ2 — What is the impact of RAG on the frequency with which smells are incorrectly identified in smell-free code?**

> The effect is model-dependent: RAG increases clean-code false positives for Qwen but reduces them for Gemini.

Figure 3 shows the proportion of smell-free files for which each model predicts at least one smell. Under **`No RAG`**, Gemini flags 79.5% of the 1,750 clean files (*i.e.*, 1,431 false-positive labels), whereas the Qwen models are much more conservative, flagging only 2–5%.

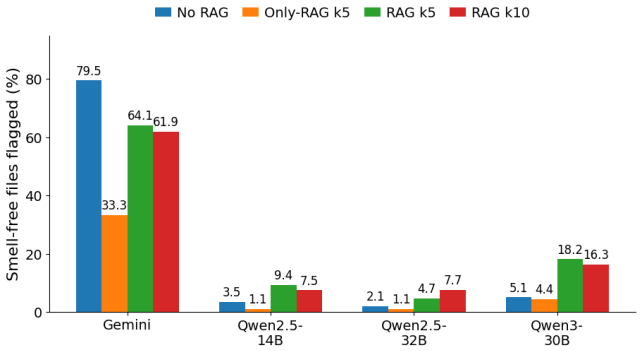


**Figure 3: Percentage of smell-free files flagged with at least one smell.**

RAG affects the models differently. For Qwen, false positives increase substantially (*e.g.*, Qwen3-Coder 30B rises from 179 to 487 labels, or from 5.1% to 18.2% of smell-free files, with **`RAG k=5`**), whereas Gemini decreases from 1,431 to 1,154 labels. Although **`Only-RAG`** produces the fewest false positives for all models, this comes at the cost of the severe recall degradation observed in RQ1.

The reason for Gemini's opposite behavior remains unclear. A possible explanation is that retrieved examples calibrate its predictions, whereas Qwen tends to overpredict smells present in the retrieval context. Supporting this hypothesis, 73.6% of all false-positive labels under RAG correspond to smells appearing among the retrieved examples for the same file, well above the expected exposure rate (14.3% for $k = 5$ and 28.6% for $k = 10$). Although this does not establish causality, it suggests that retrieved examples can bias models toward predicting the smells they contain, particularly for ambiguous smells such as *long_function*, *complex_branching*, *primitive_obsession*, and *duplicated_code*.

**RQ3 — Which types of Elixir code smells are more easily identified by LLMs, and which remain more difficult?**

> LLM performance varies across smells. The lowest-performing ones are mainly limited by low recall, with retrieval quality explaining only part of the variation.

Table 1 reports the five highest and lowest-performing smells by mean F1 across the 16 model–strategy configurations. The additional precision and recall columns show that the lowest-performing smells are not merely imprecise, they are rarely identified at all. All five lowest-performing smells have mean recall below

**Table 1: Highest and lowest-performing smells by mean per-smell metrics.**

| Smell | F1 | Prec. | Rec. | *R@10* |
|---|---|---|---|---|
| ***Five highest*** | | | | |
| data_manipulation_by_migration | 0.846 | 0.916 | 0.825 | 1.00 |
| long_parameter_list | 0.704 | 0.740 | 0.760 | 0.92 |
| agent_obsession | 0.702 | 0.614 | 0.865 | 1.00 |
| modules_with_identical_names | 0.695 | 0.996 | 0.652 | 0.04 |
| using_exceptions_for_control_flow | 0.633 | 0.704 | 0.759 | 0.48 |
| ***Five lowest*** | | | | |
| working_with_invalid_data | 0.003 | 0.003 | 0.002 | 0.00 |
| untested_polymorphic_behaviors | 0.017 | 0.060 | 0.010 | 0.52 |
| inappropriate_intimacy | 0.020 | 0.064 | 0.013 | 0.36 |
| speculative_generality | 0.022 | 0.026 | 0.019 | 0.08 |
| speculative_assumptions | 0.046 | 0.210 | 0.033 | 0.42 |

** R@10 is the retrieval Recall@10 of the embedding component.*

0.04, indicating that the models usually fail to predict these labels when they are expected.

The retrieval column shows that this behavior is only partially explained by retrieval. *working_with_invalid_data* has both near-zero model recall and zero retrieval recall, suggesting a retrieval bottleneck. However, other low-performing smells, such as *untested_polymorphic_behaviors*, *inappropriate_intimacy*, and *speculative_assumptions*, have non-zero *R@10* but still remain rarely predicted. This suggests that, for these smells, retrieving related examples is not sufficient for the model to identify the expected label.

The highest-performing smells show different patterns. Some combine high precision and recall, such as *data_manipulation_by_migration*. Others reveal more specific behavior: *agent_obsession* has high recall but lower precision, suggesting more overprediction, while *modules_with_identical_names* has almost perfect precision despite very low retrieval recall. This suggests that, for some smells, the catalog description may be sufficient, even when retrieval rarely returns examples of the expected smell.

Overall, smell-specific performance is shaped by different factors. Retrieval failure explains some cases, but not all. In others, the bottleneck appears to be the model's ability to use the available context or map the code to the correct smell definition. We leave a deeper investigation of smell-specific difficulty to future work.

### RQ4 — Does LLM performance vary across smell categories?

> Yes. Design-related smells obtain the best overall performance, while traditional smells remain the hardest category despite relatively high retrieval coverage.

Table 2 summarizes the category-level results. Design-related smells obtain the highest mean F1 (0.452), followed by low-level concerns (0.255) and traditional smells (0.222). The precision–recall split shows that all categories are limited more by recall than by precision, but in different ways. Low-level concerns have the largest precision–recall gap, suggesting that models are relatively conservative for this category: when they predict these smells, they are often correct, but many expected instances are missed.

Traditional smells show a different pattern. They have the lowest precision (0.375) despite relatively high retrieval coverage (*R@10* = 0.734). Thus, their lower performance is not primarily a retrieval problem. Instead, it suggests difficulty in mapping the analyzed code to broad smell concepts, especially for smells that often depend on thresholds or subjective judgments, such as whether a function is too long, or whether duplication is significant.

**Table 2: Performance by smell category macro-averaged over smells and over the 16 configurations.**

| Category | F1 | Prec. | Rec. | *R@10* | ΔF1 |
|---|---|---|---|---|---|
| Design-related | 0.452 | 0.565 | 0.512 | 0.807 | +0.194 |
| Low-level concerns | 0.255 | 0.521 | 0.239 | 0.424 | +0.060 |
| Traditional | 0.222 | 0.375 | 0.243 | 0.734 | +0.104 |

** ΔF1 is the average gain of RAG over No RAG.*

Design-related smells lead on both precision and recall and also benefit the most from RAG. This suggests that language-specific Elixir smells are not necessarily harder for LLMs. Rather, performance depends on how clearly the smell can be mapped from the provided code and context.

### RQ5 — Do LLMs exhibit consistent confusion patterns between specific smells in the Elixir smell catalog?

> Yes. Confusions concentrate in recurring smell pairs, and RAG changes the dominant ones.

Figure 4 shows the union of the top-5 smell confusion pairs across the four prompting strategies. A confusion occurs when the expected smell is missed and another smell is predicted for the same file. Although each strategy produces 253–350 distinct expected–predicted pairs, the errors are highly concentrated: the top five pairs account for 13.6–14.6% of all confusion events, and the top 20 for 34–37%. This is well above the uniform baseline of 1.4–2.0% for the top five pairs.

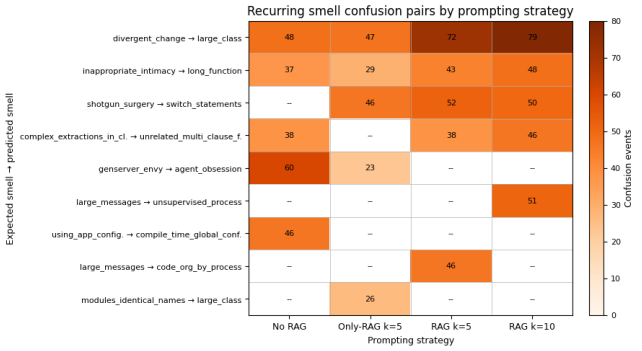


**Figure 4: Recurring confusion pairs by prompting strategy.**

Some confusions remain stable across strategies, such as *divergent_change → large_class* and *inappropriate_intimacy → long_function*. However, RAG also changes which pairs dominate: *large_messages* confusions appear only with RAG, whereas *using_app_configuration → compile_time_global_configuration* appears only under **No RAG**. Thus, RAG reshapes the confusion structure rather than simply reducing confusion errors.

### RQ6 — Do larger or newer models provide consistent gains in Elixir code smell identification?

> Partially. Larger models improve performance within the same family, but newer or stronger models do not dominate across all criteria.

Table 3 compares average performance across the four prompting strategies. Within the Qwen2.5 family, the 32B model consistently outperforms the 14B model.

**Table 3: Average model performance across the four prompting strategies.**

| Model | Macro P | Macro R | Exact Match | Jaccard |
|---|---|---|---|---|
| Gemini 3.1 Flash Lite | 0.465 | **0.539** | 0.425 | 0.460 |
| Qwen2.5-Coder 14B | 0.442 | 0.195 | 0.546 | 0.561 |
| Qwen2.5-Coder 32B | **0.567** | 0.271 | **0.582** | **0.602** |
| Qwen3-Coder 30B | 0.481 | 0.394 | 0.530 | 0.587 |

**Exact Match is computed over all files, including smell-free files.*

However, model recency does not lead to uniform gains. Qwen3-Coder 30B obtains substantially higher Macro Recall than both Qwen2.5 models, suggesting better coverage of expected smell labels. Nevertheless, it does not surpass Qwen2.5-Coder 32B in Macro Precision, Exact Match, or Mean Jaccard. Thus, the newer local model is less conservative and detects more expected smells, but the older 32B model remains stronger in overall agreement.

Gemini 3.1 Flash Lite shows the opposite trade-off. It achieves the highest Macro Recall, indicating the broadest coverage of expected smell labels. However, it also has the lowest Exact Match and Mean Jaccard. Since Exact Match is computed over all files, including smell-free files, this result is consistent with Gemini's stronger tendency to overpredict smells on clean code.

Overall, no model is universally best: Gemini favors recall, whereas Qwen2.5-Coder 32B favors precision and overall agreement. Thus, model selection depends on the desired trade-off between recall and precision.

## 5 Implications for Elixir Smell Detection Tools

Our results suggest that an LLM-based Elixir smell detector should be both smell-aware and strategy-aware, as no single prompting strategy performs equally well across all smells. For smells with poor retrieval, improving the retrieval component (*e.g.*, Elixir-specific embeddings or smell-aware retrieval) may be more effective than changing the language model alone. However, retrieval is not the only limitation: some smells remain difficult even when relevant examples are retrieved, indicating the need for richer contextual information or more discriminative prompting. Since confusion errors concentrate in recurring smell pairs, tools should expose retrieved examples, alternative labels, or brief explanations to help developers inspect ambiguous predictions rather than presenting a single definitive diagnosis.

Model selection also has practical implications. Gemini provides broader smell coverage but produces substantially more false positives on clean code, whereas Qwen2.5-Coder 32B achieves higher precision and overall agreement with the benchmark. Therefore, exploratory code review tools may benefit from higher-recall models, while IDE or CI integration should prioritize precision. More generally, predictions should be presented as recommendations rather than definitive findings, particularly because many Elixir smells involve design decisions that require human judgment. Prior work also shows that excessive false positives are a major barrier to adoption of program analysis tools [1], reinforcing the importance of conservative reporting policies.

## 6 Threats to Validity

**Dataset validity.** ExSmell-Gold is a synthetic benchmark derived from the Elixir smell catalog and may not capture the full diversity of real projects. In addition, it was validated through sampled manual inspection rather than exhaustive review. We mitigated this threat by generating the evaluation dataset independently from the retrieval corpus and performing two-author validation.

**Model and prompting validity.** Our study evaluates one proprietary model, three open-weight models, and four prompting strategies. Other models, prompts, retrieval settings, or values of $k$ may produce different results. Therefore, our conclusions are limited to the evaluated configurations.

**Evaluation validity.** Code smell identification is inherently subjective for some smells that depend on design intent or project context. To reduce this threat, we complement aggregate metrics with analyses of clean-code false positives, per-smell performance, and confusion patterns.

## 7 Conclusion and Future Work

This paper presented an empirical evaluation of LLM-based Elixir code smell identification using ExSmell-Train, ExSmell-Gold, and ExSmellBench. Our results show that RAG improves multi-label smell identification only when combined with smell descriptions, while retrieved examples alone are insufficient. Performance varies substantially across smells, categories, and models: design-related Elixir smells perform best, traditional smells remain difficult despite high retrieval coverage, and confusion errors concentrate in recurring smell pairs. No model dominates across all criteria: Gemini provides broader smell coverage, whereas Qwen2.5-Coder 32B achieves higher precision and agreement.

Future work includes evaluating real Elixir repositories with developer assessment, exploring additional models and retrieval strategies, and combining RAG with Elixir-specific static analysis and repository-level context.

## Artifact Availability

The ExSmell-Train retrieval corpus, the ExSmell-Gold evaluation benchmark, ExSmellBench, the experimental scripts, and the complete benchmark results are publicly available at https://doi.org/10.5281/zenodo.21287673, enabling the reproduction and further analysis of the reported experiments.

## Acknowledgements

This research was supported by the Brazilian funding agency CNPq through a PIBIC/CNPq grant (2025–2026). ChatGPT was used to assist in generating Figure 1, which summarizes the empirical methodology. The authors reviewed the final figure and are responsible for the paper content.

## References


[1] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 332–343. doi:10.1145/2970276.2970347

[2] Mateus José Dias and Lucas Francisco da Matta Vegi. 2026. ExSmell-Gold: Gold Standard Dataset. https://github.com/labd2m/ExSmell-Gold.

[3] Mateus José Dias and Lucas Francisco da Matta Vegi. 2026. ExSmell-Train: Dataset and Experimental Artifacts. https://github.com/labd2m/ExSmell-Train.

[4] Mateus José Dias and Lucas Francisco da Matta Vegi. 2026. ExSmellBench: A Benchmarking Tool for Evaluating LLM-Based Elixir Code Smell Detection. https://github.com/labd2m/ExSmellBench.

[5] Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code* (1 ed.). Addison-Wesley.

[6] Qi Guo, Xiaofei Xie, Shangqing Liu, Ming Hu, Xiaohong Li, and Lei Bu. 2025. Intention is All You Need: Refining Your Code from Your Intention. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 1127–1139. doi:10.1109/ICSE55347.2025.00191

[7] Saša Jurić. 2024. *Elixir in action* (3 ed.). Manning.

[8] Mahmoud Kassem, Francisco Ribeiro, and Sarah Nadi. 2026. An Automated Methodology for Generating Labeled Datasets of Semantic Errors in Code. In *Proceedings of the 3rd International Workshop on Large Language Models For Code (LLM4Code '26)*. Association for Computing Machinery, New York, NY, USA, 14–20. doi:10.1145/3786181.3788730

[9] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173. doi:10.1162/tacl_a_00638

[10] Luciana Lourdes Silva, Jânio Silva, João Eduardo Montandon, Marcus Andrade, and Marco Tulio Valente. 2024. Detecting Code Smells using ChatGPT: Initial Insights. In *18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–7.

[11] Elder Sobrinho, Andrea De Lucia, and Marcelo Maia. 2021. A systematic literature review on bad smells–5 w's: which, when, what, who, where. *IEEE Transactions on Software Engineering* 47, 1 (2021), 17–66. doi:10.1109/TSE.2018.2880977

[12] Lucas Francisco da Matta Vegi and Marco Tulio Valente. 2023. Understanding code smells in Elixir functional language. *Empirical Software Engineering* 28, 102 (2023), 1–32. doi:10.1007/s10664-023-10343-6