Effectiveness of Small and Large Language Models for PL/SQL Bad Smell Detection

Vinicius Ferreira de Sousa[1], **Cláudio de Souza Baptista**[1], **André Luiz Firmino Alves**[2], **Hugo Feitosa de Figueirêdo**[3]

[1]Universidade Federal de Campina Grande (UFCG) – Campina Grande – PB – Brazil

[2]Instituto Federal de Educação da Paraíba (IFPB) – Picuí – PB – Brazil

[3]Instituto Federal de Educação da Paraíba (IFPB) – Esperança – PB – Brazil

vinicius.sousa@ccc.ufcg.edu.br, baptista@computacao.ufcg.edu.br, {andre.alves,hugo.figueiredo}@ifpb.edu.br

Abstract. *The quality of PL/SQL code is critical for enterprise systems built on Oracle Database, yet traditional quality-assurance methods struggle to uncover nuanced code smells and complex semantic flaws. This study evaluates the effectiveness of two large language models (GPT-4o, Gemini 2.0 Flash) and two small language models (GPT-4o mini, Phi-4) for detection of bad smells in PL/SQL code. Using a uniform prompt structure, each model was tasked with identifying bad smells in a curated dataset of PL/SQL snippets. Results show that effectiveness varied significantly across models and bad smell types. These findings offer practical insights to aid in selecting and leveraging language models for PL/SQL code analysis.*

1. Introduction

The SQL/PSM standard, particularly Oracle's PL/SQL, is a critical component in many enterprise systems, allowing for the implementation of complex business logic directly within the database. This tight integration offers performance advantages [Russell et al. 2003] but also means that PL/SQL codebases are prone to bugs and "bad smells"—characteristics in the source code or database structure that, while not necessarily bugs, indicate design problems hindering maintainability, understandability, and performance [Almeida Filho et al. 2019]. The reliability of PL/SQL code is crucial for mission-critical applications, yet ensuring its quality presents significant challenges, especially in large, legacy systems where maintenance is already a considerable burden.

In a database context, the term bad smell extends beyond simple code errors to include issues like inefficient SQL queries or poorly chosen index types. Almeida Filho et al. [2019] analyzed 20 PL/SQL projects for 49 distinct types of bad smells, which were categorized into areas such as efficiency, code correction, code structure, maintenance, and readability. In practice, not all bad smells appear frequently; the analysis found that 18 of the 49 predefined smells did not occur at all in the projects examined. However, some projects can contain a very high number of these issues, with one large project in particular exhibiting 4,111 instances of bad smells.

Traditionally, maintaining PL/SQL code quality has depended on manual code reviews and static analysis tools. However, these methods have notable drawbacks. Manual reviews are labor-intensive and do not scale effectively for large codebases. Existing static analysis tools are often limited by predefined rules, can lag behind evolving coding practices and may fail to detect more nuanced, context-dependent code smells or semantic flaws [Thomson 2021].

Recent breakthroughs in Artificial Intelligence (AI), especially the advent of large language models (LLMs) and their smaller, more efficient counterparts (small language models - SLMs), present a promising new frontier for code analysis. These models have demonstrated a remarkable ability to understand both human and programming languages [Zhang et al. 2024], suggesting they could overcome the limitations of traditional methods for identifying issues in PL/SQL code. Our research is driven by the need to explore this potential, as the effectiveness of language models in the specialized context of PL/SQL remains largely unexamined, and the trade-offs between large and small models for this specific task are not yet understood.

Therefore, the primary objective of this research is to evaluate and compare the effectiveness of representative LLMs and SLMs in detecting issues and bad smells within PL/SQL code. We aim to quantify their performance, identify their strengths and weaknesses in this task, and ultimately provide insights into the practical applicability of these generative AI models as tools for improving PL/SQL code analysis and quality assurance processes within database development workflows. This investigation seeks to contribute empirical evidence to the growing field of generative AI in software engineering, specifically addressing the unique challenges and opportunities presented by SQL/PSM-based database procedural languages like PL/SQL.

The remainder of this paper is structured as follows. Section 2 presents and discusses some work related to the topic of this research. Section 3 details the methodology used to carry out the research. Section 4 focuses on the presentation and discussion of the results. Finally, Section 5 ends the paper, drawing conclusions and proposing future work.

2. Related Work

This research explores the application of both SLMs and LLMs for identifying issues and bad smells in PL/SQL code. As software complexity grows, traditional methods prove insufficient for detecting nuanced and complex code quality problems. The integration of Natural Language Processing (NLP) into software engineering presents a promising approach to enhance the accuracy and efficiency of code analysis, aiming to significantly improve software quality and maintainability [Zhang et al. 2024]. This work builds upon a growing body of literature that recognizes the potential of intelligent systems to revolutionize how code is written and refined.

Several studies underscore the transformative impact of NLP across the software development lifecycle, noting the dominance of pre-trained Transformers and LLMs [Zhang et al. 2024]. These advanced models are increasingly applied to automate complex tasks, notably in code review. AI-driven tools can analyze code, identify potential flaws, and suggest improvements with minimal human oversight [Almeida et al. 2024, Gopinath 2023]. A key advantage is their capacity for context-aware analysis, which allows for more accurate suggestions and fewer false positives than conventional static analysis tools, thereby enabling a deeper understanding of code structure and logic to uncover subtle anomalies [Almeida et al. 2024, Gopinath 2023].

To optimize performance of LLMs in code review, researchers have explored various fine-tuning techniques. Methods such as Quantized Low-Rank Adaptation (QLoRA) have shown promise in improving the generation of review comments [Haider et al. 2024]. Furthermore, augmenting prompts with semantic information, like function call graphs and code summaries, has proven beneficial [Haider et al. 2024].

The detection of code smells and anti-patterns has traditionally been the domain of static analysis tools. However, the inherent limitations of these rule-based tools have spurred the exploration of machine learning (ML) approaches [Yadav et al. 2024]. Various ML algorithms have been successfully applied to learn the characteristics of code smells from labeled datasets [Yadav et al. 2024]. Recent research has also explored the potential of prompt learning with LLMs for multi-label code smell detection [Liu et al. 2024]. Studies have shown that LLMs can identify code quality issues and code smells with competitive performance compared to static analysis tools [Holden and Kahani 2024, Lucas et al. 2024, Sheikhaei et al. 2024, Sengamedu and Zhao 2022, Silva et al. 2024]. While much research has concentrated on languages like Java and Python, there is a growing interest in applying ML and NLP to SQL and PL/SQL.

Specific research in the PL/SQL domain underscores the need for better tools. For instance, Nascimento et al. [2013] developed a static analysis tool to automate the PL/SQL code review, demonstrating significant benefits in real-world applications. Later, an exploratory analysis by Almeida Filho et al. [2019] of public PL/SQL repositories revealed that certain bad smells tend to co-occur, reinforcing the importance of early detection to minimize future refactoring efforts. These studies highlight a recognized need for robust, automated quality assurance in PL/SQL development.

Comparative analyses of SLMs and LLMs in software development reveal a trade-off between computational resources and performance [Hassid et al. 2024]. LLMs generally excel in complex language understanding and generation but require significant computational power, whereas SLMs offer efficiency advantages and can achieve comparable performance on specific tasks [Sun et al. 2024].

Despite promising advancements, the application of language models to code analysis has limitations, including hallucination, difficulties with complex reasoning and processing long code sequences [Fang et al. 2024]. Nevertheless, a clear research gap exists regarding the specific, comparative assessment of both small and large language models for detecting issues and bad smells within PL/SQL code. This research aims to fill that void by providing a focused evaluation of different models in this task. The findings are expected to offer valuable insights into the suitability of these models for this widely used database programming language, potentially paving the way for more effective tools to improve the quality and maintainability of PL/SQL programs.

3. Methodology

This research employed an empirical, comparative evaluation methodology to assess the effectiveness of selected large and small language models in detecting issues and bad smells in PL/SQL code. The core approach involves presenting curated PL/SQL code snippets containing known issues to different models and analyzing their ability to identify these problems accurately. Figure 1 shows an overview of the methodology, the steps of which are detailed in the following subsections.

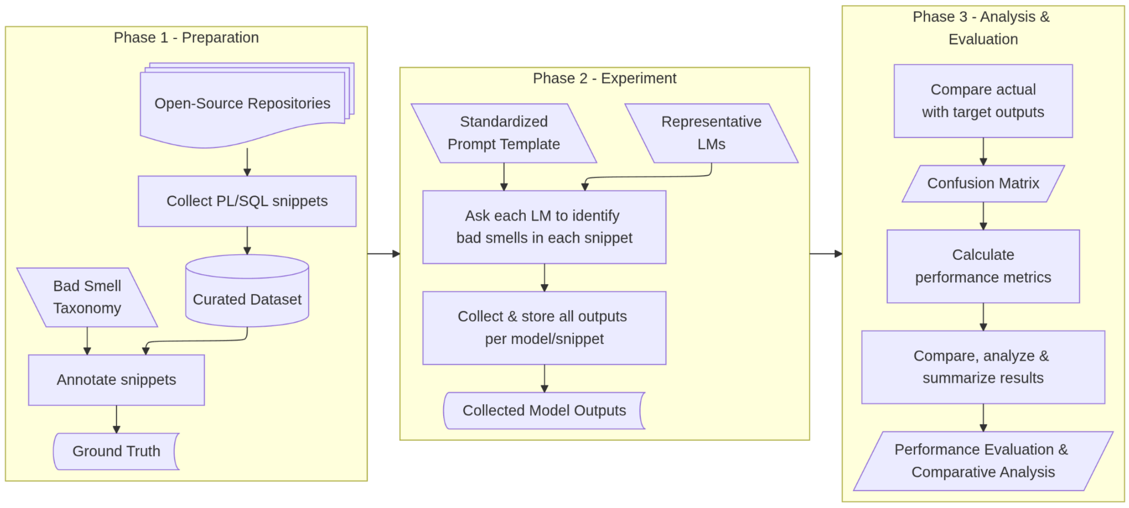
Phase 1 - Preparation
Open-Source Repositories
Collect PL/SQL snippets
Curated Dataset
Bad Smell Taxonomy
Annotate snippets
Ground Truth
Phase 2 - Experiment
Standardized Prompt Template
Representative LMs
Ask each LM to identify bad smells in each snippet
Collect & store all outputs per model/snippet
Collected Model Outputs
Phase 3 - Analysis & Evaluation
Compare actual with target outputs
Confusion Matrix
Calculate performance metrics
Compare, analyze & summarize results
Performance Evaluation & Comparative Analysis

Figure 1. Research methodology overview

3.1. Model Selection

To represent the state-of-the-art (at the time of this study's completion, April 2025) and explore the trade-offs between LLMs (more comprehensive) and SLMs (more efficient), the following language models were selected for evaluation:

- **GPT-4o** (OpenAI): A large multimodal model known for its strong reasoning and code understanding capabilities across various languages [OpenAI 2024b].
- **GPT-4o mini** (OpenAI): A smaller, more cost-effective variant of GPT-4o, an SLM designed for faster responses and efficiency, allowing us to assess the performance trade-offs within the same model family [OpenAI 2024a].
- **Gemini 2.0 Flash** (Google): An LLM optimized for speed and efficiency, a robust, fast model suitable for latency-sensitive or high-volume, high-frequency tasks at scale, capable of multimodal generation, and featuring a 1 million token context window [Google DeepMind 2024].
- **Phi-4** (Microsoft): An open SLM specialized in complex reasoning that offers high-quality responses at a small size, with scores superior or comparable to the aforementioned models and other competitors on various benchmarks, surpassing them in the parameter count to performance ratio [Abdin et al. 2024].

The choices were guided by a goal to evaluate a diverse and representative sample of models available as of April 2025, focusing on three criteria: (1) state-of-the-art performance, (2) architectural variety and model size (LLM vs. SLM), and (3) different development philosophies (proprietary vs. open) and providers (OpenAI, Google, Microsoft). Our chosen models reflect these criteria:

- LLMs: We selected GPT-4o for its top-tier reasoning and code understanding and Gemini 2.0 Flash for its focus on speed and efficiency at scale as well as great performance on code-focused benchmarks.

- SLMs: We chose Phi-4 as a leading open-source model praised for its high performance-to-size ratio and GPT-4o mini to directly assess the performance trade-offs within a single model family (compared to GPT-4o).

This multi-faceted rationale ensured our study evaluated a diverse representation of the capabilities and resource requirements available at the time of this writing, allowing for a more reliable set of insights.

Access to the models was obtained via their standard interfaces publicly available on the web—except for Phi-4, which was tested via the Chatbot Arena[1] platform, with parameters temperature=0.7, top-P=1, and max. output tokens set to 2048—during the experimental period, between March and April 2025.

3.2. Dataset Curation

A dataset of PL/SQL code snippets was purpose-built for this study. The dataset was compiled to include a variety of issues and bad smells relevant to PL/SQL development. These snippets were extracted from publicly available code repositories used for testing the tool ZPA[2], a free and open-source PL/SQL static analyzer. A stratified sample was assembled in order to hold code examples that could be used to compare the performance of language models with the performance of a static analyzer, hence snippets that are used to test the ZPA tool were selected. The stratified sampling considered several features so as to ensure structural diversity among the examples and to allow a fairer comparison between the language models and the ZPA tool in future work, since the selected snippets are part of the set used to validate that tool.

Each snippet in the dataset was carefully annotated to establish a ground truth. This annotation, performed manually by researchers (experienced in PL/SQL code development and quality assessment) considering the issues reported by the tool that they judged to be true positives, identified and cataloged all known issues present within the code. The types of issues targeted for detection are listed in Table 1, which also includes their description and explanation (why they are issues). All of them are bad smells, except for IEW and PMO.

Only types of issues whose identification does not depend on context external to the code snippet were selected, thus ensuring the validity of the experiment even in the absence of metadata or information about the database schema. The PL/SQL code snippets do not belong to the same database schema, which contributes to the diversity of the dataset. The description of these schemas is not necessary, since the issues selected can be identified regardless of the structure of the underlying database.

The dataset comprised 44 snippets, varying in length (5–148 lines of code) and complexity (low, medium, high), ensuring a range of challenges for the models. The average length was approximately 27 lines (the snippets had 1186 lines altogether), the median length was 15. Regarding complexity, 3 snippets were high complexity, 12 had medium complexity and the remaining 29 were classified as low complexity. Furthermore, the ground truth contained a total of 61 distinct instances of issues across

[1] https://lmarena.ai
[2] https://zpa.felipebz.com/

the eight categories. The distribution of instances for each issue type is as follows: there were 13 instances of CDO, 12 instances of PMO, 10 of UVP, 7 of USA, 7 of CAT, 5 of QEH, 4 of ECB, and 3 instances of IEW.

Table 1. Definition of targeted issue types

No.	ID	Description	Explanation (why it is an issue)
1	UVP	Unused local variable or unused parameter.	It indicates the presence of dead or incomplete code, which increases cognitive load, reduces code clarity and impedes maintainability. Such artifacts can mislead developers regarding the code's intended behavior.
2	CDO	Call to procedure from package `DBMS_OUTPUT`.	It is an indicator that the code is not production-ready. The presence of `DBMS_OUTPUT` calls in a final codebase may indicate that development artifacts were not properly removed. For application logging or instrumentation, a logging mechanism should be preferred.
3	IEW	`IF` ... `EXIT` instead of `EXIT WHEN` to exit from a loop.	The `EXIT WHEN` construct clearly conveys the loop's termination logic in a single, declarative line, aligning with structured programming principles and PL/SQL idioms. In contrast, using an `IF` statement for that obscures the loop's exit condition, increasing the cognitive load and raising the risk of errors.
4	ECB	Empty code block.	It indicates incomplete or missing logic that the developer intended to implement but forgot or omitted, which can result in unintended behavior. It can mislead maintainers into believing that the absence of code is intentional, potentially masking bugs or critical logic gaps.
5	PMO	Parameter mode omission.	It obscures the intended data flow and contractual interface of the routine, increases the risk of unintended side-effects or misuse by inexperienced developers and complicates maintenance, as peers must infer behavior rather than rely on explicit declarations.
6	USA	Use of `SELECT *` (asterisk wildcard).	This retrieves all columns from a table indiscriminately, which can lead to reduced query performance and may inadvertently expose sensitive or deprecated columns. It also introduces maintainability issues, as changes to the underlying table schema may cause unintended side effects in dependent code.
7	CAT	`COMMIT`/ `ROLLBACK` in a non-autonomous transaction.	It undermines transaction management at the application level. Transaction demarcation should be managed by the caller or a dedicated transaction controller rather than by individual PL/SQL units, otherwise one risks prematurely finalizing work, thereby introducing unwanted side-effects.
8	QEH	Query execution without exception handling.	A `SELECT INTO` statement without an associated exception handler leaves the program vulnerable to unhandled runtime errors. Without exception handling, these conditions propagate as unanticipated runtime exceptions, undermining the code's robustness and maintainability.

3.3. Experimental Procedure

Each PL/SQL snippet from the curated dataset was presented individually to each of the selected language models. A standardized prompt, written in English aiming for better results and designed according to prompt engineering principles, was used across all models and snippets to ensure consistency in task instruction. The prompt text had a fixed part and a variable part, the placeholder for the snippet, where each code snippet was interpolated. The prompt structure/template is shown in Figure 2.

You are an AI assistant specialized in PL/SQL code analysis and linting.

Consider the PL/SQL code smells listed below:

1. Unused local variable or unused parameter.
2. Call to procedure from package `DBMS_OUTPUT`.
3. Use of an `IF` statement instead of `EXIT WHEN` to exit from a loop.
4. Empty code block (`BEGIN NULL; END;`).
5. Parameter mode omission (parameter declaration without explicit `IN`, `OUT`, or `IN OUT`).
6. Use of `SELECT *` (projection of all columns).
7. `COMMIT` or `ROLLBACK` in a non-autonomous transaction.
8. `SELECT INTO` statement without an associated exception handler covering potential query exceptions (e.g. `NO_DATA_FOUND`, `TOO_MANY_ROWS`).

Analyze the provided PL/SQL code snippet and check for those code smells.

If any smells are detected, present them as a list where each item matches this format:

- Location: <line(s) where the smell occurs>
- Issue: <smell number from the list above>

If none of the specified smells are identified, respond with this sentence:
"No smell detected."

Here is the PL/SQL code snippet:

```
{plsql_code_snippet}
```

Figure 2. Prompt template used to instruct the language models to detect PL/SQL smells

The prompt was meticulously designed according to established prompt engineering principles. Specifically, it incorporates:

- **Role-Playing**: The prompt assigns a specific role to the model ("You are an AI assistant specialized in PL/SQL code analysis and linting."). This focuses the model on the specific task.
- **Clear and Detailed Instructions**: It provides an explicit, numbered list of the eight issues to be detected, with their descriptions.
- **Structured Output Format**: The prompt strictly defines the output format, requiring a specific template for reporting detected smells and a precise fallback sentence ("No smell detected.") if none are found. This was crucial for standardizing the outputs for our analysis.

- **Standardization**: The exact same prompt template was used for every model and every code snippet, ensuring that any performance variation could be attributed to the models themselves, not the instructions.

This prompt design aimed to control the experiment and yield structured, comparable data. The complete textual output generated by each model was captured verbatim for subsequent analysis. The models were queried sequentially for each snippet to avoid cross-contamination or learning effects during the experiment. Default generation parameter values (e.g., temperature, top-p) were used for each model, so as to reflect typical usage patterns. All code snippets were presented to the models in their original form, without any pre-processing, in order to simulate realistic scenarios.

3.4. Analysis and Metrics

The core of the analysis involved comparing the output generated by each model against the predefined ground truth for each code snippet. This comparison was performed manually by the researchers. For each potential issue reported by a model, we determined whether it corresponded to a known issue in the ground truth. This allowed us to classify each reported issue as follows:

- true positive (TP), the model correctly identified an issue present in the ground truth for that snippet;
- false positive (FP), the model identified an issue that was not present in the ground truth (i.e., an erroneous detection, an alert that is not actually an issue);
- false negative (FN), the model failed to identify an issue present in the ground truth for that snippet;
- true negative (TN), absence of an issue correctly assumed and issue not reported, the model did not detect an issue and there is no issue listed in the ground truth for that snippet.

Based on these classifications, aggregated across all snippets, we calculated the following standard performance metrics—widely used in machine learning for evaluating the performance of predictive classification models [Opitz 2024]— for each model, both overall (micro averages) and per issue type:

- **precision**, the ratio of actual issues (correctly identified) to all reported issues, given by Eq. 1;

$$Precision = \frac{TP}{TP+FP} \tag{1}$$

- **recall**, the TP detection rate, the ratio of correctly identified issues to all actual issues present in the ground truth, expressed as Eq. 2;

$$Recall = \frac{TP}{TP+FN} \tag{2}$$

- **F1-score**, the harmonic mean of precision and recall, providing a single measure that balances both aspects, calculated by Eq. 3.

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision+Recall} \tag{3}$$

To promote the reproducibility of this research as well as the auditing of its results, the data yielded by the preparation and experiments are publicly available on a webpage[3], including the definitions of the targeted issue types, the dataset, the ground truth, the collection of model outputs, and the raw experimental results.

4. Results

This Section presents the empirical results derived from evaluating the performance of the four selected language models in detecting specific issues within the curated dataset of 44 PL/SQL code snippets. The evaluation employed the metrics mentioned in Subsection 3.4, calculated by comparing model outputs against a manually verified ground truth containing 61 distinct issue instances across 8 categories.

4.1 Overall Model Performance

Table 2 summarizes the aggregate performance of each model across all issue types and code snippets. It presents the total counts of TPs, FPs, and FNs, along with the overall precision, recall, and F1-score per model (calculated as defined in Subsection 3.4).

Table 2. Overall model performance comparison

Model	TP	FP	FN	Precision	Recall	F1-Score
GPT-4o	57	27	4	0.68	**0.93**	**0.79**
Phi-4	55	24	6	**0.70**	0.90	0.79
Gemini 2.0 Flash	42	19	19	0.69	0.69	0.69
GPT-4o mini	45	35	16	0.56	0.74	0.64

Based on the overall F1-score, GPT-4o and Phi-4 demonstrated the highest effectiveness (0.79, a tie for first place), followed by Gemini 2.0 Flash (0.69), and lastly GPT-4o mini (0.64). GPT-4o achieved the highest recall (0.93), to which Phi-4 was second by a very small margin, indicating they both successfully identified the largest proportion of actual issues present in the dataset, albeit at the cost of moderate precision (0.68 and 0.70 respectively) due to the number of false positives being much greater than false negatives. Precision being significantly lower than recall for the two top-performing models and GPT-4o mini indicates that they reported many more absent issues (false positives) than missed actual ones (false negatives). Phi-4, Gemini 2.0 Flash, and GPT-4o, ranked in this order by difference of only a hundredth, exhibited similar precision. Gemini 2.0 Flash presented a more balanced profile with identical recall and precision; although it exhibited the weakest recall and the greatest number of false negatives, its precision was as high as the two top-performing models and it generated the fewest false positives, ranking third/second-to-last. Alongside Gemini 2.0 Flash, GPT-4o mini lagged behind, showing the lowest precision (0.56) with the highest number of false positives (35). Although it exhibited less false negatives and a slightly better recall compared to Gemini 2.0 Flash, it had the lowest F1-score, hence ranking as the worst-performing, least-effective of the four models.

4.2 Performance by Issue Type

To understand model performance nuances, Table 3 presents the F1-scores achieved by each model for the individual issue types. Performance varied considerably across issue

[3] https://tinyurl.com/4hxndke3

types and models. The second column (Support) exhibits the number of occurrences of each issue type in the ground truth.

Table 3. Model F1-scores per issue type

Issue Type	Support	GPT-4o	Phi-4	Gemini 2.0 Flash	GPT-4o mini
UVP	10	0.50	**0.52**	0.48	0.33
CDO	13	**1.00**	0.96	0.92	0.92
IEW	3	**1.00**	0.75	0.80	0.55
ECB	4	0.67	**0.89**	0.67	0.53
PMO	12	0.69	**0.75**	0.57	0.42
USA	7	**1.00**	0.93	0.77	0.71
CAT	7	**0.93**	0.88	0.71	0.88
QEH	5	**0.91**	0.71	0.75	0.71

High Performance Areas. GPT-4o led in detection of most issue types, followed by Phi-4. All models performed exceptionally well on detecting CDO, with GPT-4o achieving a perfect score and the others reaching an F1 above 0.90. GPT-4o also perfectly detected IEW and USA, in addition to reaching an excellent F1 for CAT and QEH. It achieved the highest F1-score for 5 issue types, and second highest for the remaining 3. Phi-4 in turn demonstrated excellent performance on CDO, ECB, and USA, with an F1 close to 0.90, as well as outperformed the other models on UVP, ECB, and PMO. Detection of CAT was also strong for GPT-4o, Phi-4, and GPT-4o mini (F1 close to 0.90), while Gemini 2.0 Flash was not as strong but still was good (F1 > 0.70).

Moderate Performance Areas. The models showed moderate success in detection of specific issue types, with F1-scores between 0.70 and 0.80. Gemini 2.0 Flash and GPT-4o mini were quite good at detecting USA. On the other hand, IEW was reasonably well detected by Gemini 2.0 Flash and Phi-4, which in turn led in PMO detection, with the only F1 above 0.70 for that issue type. Also, for those three models it was not so easy to detect QEH as it was for GPT-4o, but they performed moderately well on that, with similar F1-scores.

Challenging Areas. The detection of UVP proved difficult for all models, primarily due to poor precision (many false positives, most of the wrong detections belong to that category), resulting in low F1-scores ranging from 0.33 to 0.52. Performance on PMO was inconsistent, with only Phi-4 good, while others struggled, especially GPT-4o mini, which also performed poorly on detecting IEW. Similarly, ECB detection was strong only for Phi-4, with other models achieving low F1-scores (0.53–0.67).

4.3 Qualitative Analysis

Beyond the quantitative metrics, a qualitative analysis of the models' outputs reveals critical nuances in their behavior, offering insights into their strengths, weaknesses, and the underlying reasons for their performance variations.

A consistent observation across all evaluated models is the difficulty in pinpointing the exact line number where an issue occurs. In most cases, the models identified a line in close proximity to the problematic one but failed to specify the precise location. This imprecision could hinder a developer's ability to quickly locate and rectify the issue. This suggests that while the models can infer the general area of an issue, their token-level mapping to specific line numbers is not consistently accurate.

Despite this limitation in localization, the models demonstrated a strong ability to identify the relevant code segments associated with a potential issue. Without explicit guidance, they were capable of isolating the lines that could be affected by a given issue, indicating a robust inferential capacity and a clear comprehension of PL/SQL and the specified issue types. This suggests that the models' errors may not stem from the lack of such understanding but rather from more subtle factors, such as erroneous generalizations, misinterpretations of the issue definitions in specific code contexts, or a misunderstanding of the analyzed snippet itself. The inferior performance in detecting certain issues like UVP and PMO could be attributed to ambiguities in interpreting their definitions or a sensitivity to nuances in the code that leads to confusion.

A noteworthy behavior observed in all models except Gemini 2.0 Flash was a form of self-correction. GPT-4o, in particular, frequently exhibited this trait. The model would initially report a potential issue, but in the process of generating its justification, it would recognize its own error, identify the detection as a false positive, and retract the claim. The entire chain of thought, including the refutation, was preserved in the output, offering a transparent view into its reasoning.

All models adhered strictly to the constraint of focusing only on the eight specified issue types, not reporting any out-of-scope problems. However, adherence to the required output format varied. Gemini 2.0 Flash was the most compliant, consistently producing outputs that precisely matched the requested format. Conversely, it was also one of the least accurate models. Phi-4, one of the top performers, consistently produced more verbose outputs, providing justifications for its detections. GPT-4o mini exhibited similar verbosity. GPT-4o’s behavior was inconsistent; it sometimes generated concise, strictly formatted responses and at other times included additional, unsolicited explanations.

The fact that one of the best-performing models (Phi-4) consistently provided justifications, combined with the observed instances of self-correction during the justification process, suggests a promising direction for improving performance. It indicates that prompting the models to explain their reasoning—requiring them to articulate why they believe an issue is present—could act as a mechanism for internal validation, potentially reducing false positives and enhancing overall detection accuracy.

4.4 Summary of Findings

The results indicate that while modern language models can detect a significant proportion of issues and bad smells in PL/SQL code, their effectiveness varies based on the model architecture and the specific type of issue. GPT-4o and Phi-4 demonstrated the best overall performance, primarily due to the former’s superior recall across most categories and the latter’s superior precision across the most challenging ones (UVP, ECB, and PMO). Phi-4 offered higher precision but slightly less strong recall than GPT-4o. Gemini 2.0 Flash provided a balanced profile but lagged behind GPT-4o and Phi-4. GPT-4o mini exhibited the lowest overall effectiveness, hampered by both lower recall and the lowest precision. Notably, accurately identifying unused variables/parameters (UVP), omissions of parameter mode (PMO), and empty blocks (ECB) posed challenges for most models, leading to a high rate of false positives. Conversely, detecting straightforward patterns like calls to a specific package (CDO) or the use of the asterisk wildcard in the projection list of a query (USA) was handled effectively by the top-performing models.

5. Conclusion

This research evaluated the effectiveness of four language models—GPT-4o, Phi-4, Gemini 2.0 Flash, and GPT-4o mini—for detecting bad smells and issues in PL/SQL code. The results indicate that GPT-4o and Phi-4 were the most effective, with the former achieving the highest recall and the latter the highest precision. Gemini 2.0 Flash provided a balanced, moderate performance while GPT-4o mini was the least effective. Notably, the performance of all models varied significantly depending on the issue type, with simpler ones being more readily detected. Additionally, the strong performance of Phi-4 suggests that SLMs can be viable alternatives to LLMs for this task.

These findings evince the ability of language models to identify issues in PL/SQL code with minimal context, and suggest that they hold promise as assistive tools for PL/SQL code quality assurance, augmenting and complementing static analysis and manual reviews, particularly for issues that are challenging for existing tools to identify. However, the choice of model involves a trade-off between recall and precision of issue detection. Critically, no model achieved perfect accuracy, highlighting that human oversight remains essential for validating the models' outputs.

Future research should expand upon these findings by conducting statistical analyses of model performance on larger, more diverse datasets from real-world codebases to ensure generalizability. A direct benchmark comparison against established static analysis tools like ZPA, focusing on complex issues, is crucial to quantify the relative benefits of language models. Further investigation is also needed to understand how additional context, code complexity and length affect detection accuracy. Finally, future work should explore the potential of these models for refactoring PL/SQL code, with a focus on efficiency improvements. These next steps are vital for fully understanding the applicability and limitations of language models in the automation of PL/SQL code quality assurance.

Acknowledgements

The second author would like to express his gratitude to the Brazilian National Council for Scientific and Technological Development (CNPq) for the research scholarship that supported this work.

References

Abdin, M., et al. (2024). Phi-4 technical report. arXiv:2412.08905.

Almeida, Y., et al. (2024). AICodeReview: Advancing code quality with AI-enhanced reviews. *SoftwareX, 26*(101677).

Almeida Filho, F. G. de, et al. (2019). Prevalence of bad smells in PL/SQL projects. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 116–121. IEEE.

Fang, C., et al. (2024). Large language models for code analysis: Do LLMs really do their job?. In *33rd USENIX Security Symposium (USENIX Security 24)* (pp. 829–846). USENIX Association.

Google DeepMind (2024). Introducing Gemini 2.0: our new AI model for the agentic era. (11 December 2024). Retrieved April 17, 2025 from `https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/`.

Gopinath, K. (2023). Exploring the role of large language models in automated code review and software quality enhancement. *International Journal of Innovative Research in Science, Engineering and Technology*, 12(9):11428–11438.

Haider, M. A., Mostofa, A. B., Mosaddek, S. S. B., Iqbal, A., and Ahmed, T. (2024). Prompting and fine-tuning large language models for automated code review comment generation. arXiv:2411.10129.

Hassid, M., Remez, T., Gehring, J., Schwartz, R., and Adi, Y. (2024). The larger the better? Improved LLM code-generation via budget reallocation. arXiv:2404.00725.

Holden, D. and Kahani, N. (2024). Code linting using language models. arXiv:2406.19508.

Liu, H., Zhang, Y., Saikrishna, V., Tian, Q., and Zheng, K. (2024). Prompt learning for multi-label code smell detection: A promising approach. arXiv:2402.10398.

Lucas, K., Gheyi, R., Soares, E., Ribeiro, M., and Machado, I. (2024). Evaluating large language models in detecting test smells. arXiv:2407.19261.

Nascimento, D., Pires, C. E., and Massoni, T. (2013). PL/SQL Advisor: uma ferramenta baseada em análise estática para sugerir melhorias para procedimentos armazenados. In *Anais do IX Simpósio Brasileiro de Sistemas de Informação*, pages 343–354, Porto Alegre, RS, Brazil. SBC.

OpenAI (2024). GPT-4o mini: advancing cost-efficient intelligence. (18 July 2024). Retrieved April 17, 2025 from `https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/`.

OpenAI. (2024). GPT-4o system card. arXiv:2410.21276.

Opitz, J. (2024). A closer look at classification evaluation metrics and a critical reflection of common evaluation practice. *Transactions of the Association for Computational Linguistics,* 12:820–836.

Russell, J., et al. (2003). *PL/SQL User's Guide and Reference, 10g Release 1 (10.1)*, chapter 1 Overview of PL/SQL. Oracle Corporation, Redwood City, CA, USA.

Sengamedu, S. and Zhao, H. (2022). Neural language models for code quality identification. In *Proceedings of the 6th International Workshop on Machine Learning Techniques for Software Quality Evaluation*, MaLTeSQuE 2022, pages 5–10, New York, NY, USA. Association for Computing Machinery.

Sheikhaei, M. S., Tian, Y., Wang, S., and Xu, B. (2024). An empirical study on the effectiveness of large language models for SATD identification and classification. *Empirical Software Engineering*, 29(159).

Silva, L. L., Silva, J. R. da, Montandon, J. E., Andrade, M., and Valente, M. T. (2024). Detecting code smells using ChatGPT: Initial insights. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '24, pages 400–406, New York, NY, USA. Association for Computing Machinery.

Sun, Z., et al. (2024). Enhancing code generation performance of smaller models by distilling the reasoning ability of LLMs. arXiv:2403.13271.

Thomson, P. (2021). Static analysis: An introduction: The fundamental challenge of software engineering is one of complexity. *Queue*, 19(4), 29–41.

Yadav, P. S., Rao, R. S., Mishra, A., and Gupta, M. (2024). Machine learning-based methods for code smell detection: A survey. *Applied Sciences*, 14(14):6149.

Zhang, Z., et al. (2024). Unifying the perspectives of NLP and software engineering: A survey on language models for code. arXiv:2311.07989.