Scylla: A Unified Tool for Code Smell Classification in Python

Igor Soares de Oliveira
Department of Informatics, PUC-Rio
Rio de Janeiro, Brazil
isoliveira@inf.puc-rio.br

Alexandre Andrade
Department of Informatics, PUC-Rio
Rio de Janeiro, Brazil
aandrade@inf.puc-rio.br

Rayssa Athayde
Department of Informatics, PUC-Rio
Rio de Janeiro, Brazil
rayssa@aise.inf.puc-rio.br

Eduardo Nadai
Department of Informatics, PUC-Rio
Rio de Janeiro, Brazil
edunadai@aise.inf.puc-rio.br

Juliana Alves Pereira
Department of Informatics, PUC-Rio
Rio de Janeiro, Brazil
juliana@inf.puc-rio.br

Abstract

Code smells are structural indicators of poor design that can compromise software maintainability, readability, and scalability. Their detection and mitigation are essential to reduce technical debt and ensure the long-term quality of systems. Traditional approaches rely on static analysis and rule-based tools, which often lack contextual understanding and adaptability to evolving codebases. This paper presents `Scylla`, a unified tool for code smell classification in Python that unifies multiple detection approaches with a single environment: heuristic analysis based on Abstract Syntax Tree (AST), Machine Learning (ML), Deep Learning (DL), and Language Models (LMs). `Scylla` adopts an event-driven microservice architecture composed of three main components: (1) an API responsible for coordinating and processing classification tasks asynchronously, persisting AST-based heuristic, ML-, and DL-based results in a PostgreSQL database; (2) a worker service that executes SLM-based classifications via Ollama and stores results in a PostgreSQL database; and (3) a web application providing an interactive interface for code submission and classification. `Scylla`'s modular design ensures scalability, extensibility, and fault tolerance, supporting future integration of new code smells, models, and programming languages. Experimental results demonstrate the potential of `Scylla` as a unified and extensible framework for automated software quality assessment and comparative research on code smell detection techniques.

Demo video: https://youtu.be/C3YnA-Vyqc0

Keywords

Software Quality, Code Smells, Python, Heuristic, Language Models, Learning Models

1 Introduction

Code smells are structural or design characteristics in software systems that suggest potential quality issues and may increase maintenance complexity. Initially introduced by Fowler et al. [10], they represent symptoms of violations of software development principles—such as excessive coupling, low cohesion, or redundant functionality—that hinder program comprehension and contribute to higher maintenance costs and defect risks. Moreover, code smells accumulate technical debt, elevate the likelihood of failures, and reduce the productivity of development teams [17]. Refactoring—the process of restructuring existing code without changing its external behavior—is the primary strategy to address these problems. However, its effectiveness strongly depends on the accurate and timely identification of code smells. Manual inspection of large codebases to identify code smells is time-consuming, error-prone, and non-scalable, particularly as modern software projects grow in size and complexity. Consequently, there is a growing need for automated, reliable detection tools that systematically analyze code and provide consistent feedback to support refactoring decisions and continuous quality improvement.

Despite Python's growing popularity, the detection of code smells in Python remains underexplored. Existing tools, such as Pylint [31] and Pyflakes [25], focus mainly on syntactic errors and stylistic inconsistencies, offering limited insight into design-level or structural smells. Research-oriented metric-based tools, such as PySmell [5] and Dpy [3], while effective at identifying threshold violations, often lack contextual understanding, failing to capture the subtle semantic and structural relationships that characterize more complex code smells. Moreover, they are either discontinued or not openly available, leaving a gap in reliable, actively maintained, and up-to-date solutions. Furthermore, none of the existing tools provides a unified environment that supports multiple AI-driven approaches to code smell detection, such as those based on Machine Learning (ML), Deep Learning (DL), or Language Models (LMs). This limitation highlights the need for modern, extensible, and intelligent tools capable of leveraging both analytical metrics and AI-based reasoning to enhance the accuracy and interpretability of code smell detection in Python.

To address these limitations, we propose `Scylla`[1], an open-source web-based tool for detecting code smells in Python. Unlike existing tools that typically focus on a single detection approach, `Scylla` integrates multiple AI-driven and analytical approaches, including Abstract Syntax Tree (AST)-based heuristic analysis, ML, DL, and LMs. We evaluated `Scylla` through a user-centered experimental study to assess users' overall experience and task effectiveness. Our evaluation captures the impact of `Scylla` across varying levels of analytical experience, focusing on how well it assists navigation, classification, and interpretation of code smells during the review process.

Our main contributions are threefold: (1) We present `Scylla`, a unified tool for classifying code smells in Python using four

[1] `Scylla` takes its name from a famous creature in Greek mythology. Like the mythical guardian lurking in treacherous waters, `Scylla` symbolizes vigilance against hidden dangers (*i.e.* code smells) that can hinder code comprehension. The name reflects the tool's purpose to detect and expose such threats before they "devour" code quality.

approaches—AST-based heuristics, ML, DL, and LMs. (2) We introduce an extensible architecture that supports a wide range of code smell types and designed to accommodate future integration of additional approaches, models, and programming languages. (3) We provide empirical evidence, based on controlled experiments with Python developers, that `Scylla` assists code review activities by streamlining the classification of diverse code smell types, enabling comparisons across detection approaches, and helping developers better interpret and reason about potential quality issues in the analyzed code.

2 Related Work

The detection of code smells has traditionally relied on AST-based tools, but recent years have seen growing interest in ML, DL, and LLM-based approaches. An early study by Mäntylä and Lassenius [16] proposed a taxonomy of bad smells by grouping related smells into higher-level categories and presented empirical evidence of their role in developers' subjective evaluations of code quality. This study helped shape subsequent discussions on bad smell classification. Later, Lanza and Marinescu [14] formalized the use of structural metrics and detection strategies for identifying smells such as Long Method and Long Parameter List.

In the Python ecosystem, several semantic-based and AST-based heuristic tools are widely used. Pylint [31] and Flake8 [24] perform stylistic and structural checks to identify dead code, style violations, and maintainability issues. Similarly, MLpylint [12] was designed to detect ML-specific code smells. The DPy [3], PySmell [5], and PyExamine [29] use AST-based heuristic analysis to detect code smells through structural metrics and semantic rules. Several studies employ ML and DL models to automatically classify code smells by learning patterns from source code representations (such as ASTs, tokens, and code embeddings) and using these learned features to distinguish smelly from non-smelly code. Azeem et al. [2] presented a systematic review of supervised algorithms, including Decision Trees, Random Forests, and SVMs, showing that ensemble and rule-based models achieve high performance. For instance, Rao et al. [27] used techniques like XGBoost and Bagging combined with SMOTE to handle class imbalance, achieving strong predictive performance for Python code smell detection. Previous studies [13, 15] use DL for smell detection, leveraging embeddings derived from syntactic structures. Recently, studies [19, 22] demonstrated that LMs can accurately identify smells, adapting dynamically to different code contexts. These learning-driven approaches represent a shift from purely static, rule-based detection to more context-aware and data-driven analysis, enabling the discovery of subtle and domain-specific code smells. In this context, `Scylla` aims to integrate the strengths of all these approaches–AST-based heuristic analysis, ML and DL, and LM-based contextual reasoning–into a modular and scalable framework for code smell detection in Python.

3 Scylla

`Scylla` was developed following the principles of event-driven microservice architecture. This architectural approach allows services to operate independently while communicating asynchronously through well-defined events, ensuring robustness and fault tolerance. It also facilitates the incremental incorporation of new types of code smells, including those from different programming languages, with minimal impact on existing components. Currently, `Scylla` supports the detection of multiple code smell types, as illustrated in Table 1 and Figure 2 illustrates the architecture of `Scylla`, highlighting its main services and data flow. Next, we describe each of these components, explaining their roles, interactions, and how they collectively support multi-approach code smell classification within a unified tool.

(1) *Web:* An interactive web application was developed to facilitate the classification of Python code smells through multiple detection approaches, as illustrated in Figure 1, including AST-based heuristics, ML, DL, and LMs. Through the interface, users can upload one or more source code files, define the analysis granularity (*e.g.*, class or method level), select the target code smell, and choose the desired detection approach or model. For LM-based analyses, the platform additionally supports the selection of different LMs, as illustrated in Table 2, prompting strategies, including zero-shot and chain-of-thought prompting [26, 34], and optional inclusion of software metrics within prompts, enabling flexible experimentation across distinct refactoring and detection configurations.

In addition to the execution workflow, the platform provides a task management interface that allows users to track previously executed analyses, inspect detailed detection results directly within the application, export classification outputs in CSV format, and provide feedback regarding the detected code smells. To improve transparency and reproducibility, the system also offers detailed visual reports that describes the detection outcomes generated by each approach.

The web application further includes an analytical dashboard designed to support exploratory analysis of the generated results. Through this dashboard, users can visualize information related to tool usage, frequency of executed analyses, distribution of detected code smells, success rates, and the utilization of different detection approaches. Moreover, `Scylla` provides a concordance analysis module capable of comparing the output generated by different detection approaches, enabling the identification of agreement and conflict scenarios among heuristic, ML, DL, and LM-based detections. The web application was implemented using widely adopted technologies, JavaScript [9], CSS [20], and HTML [30].

(2) *code-extractor.api:* It serves as the central processing component within the proposed architecture. *code-extractor.api* is responsible for orchestrating the reception of requests from the *Web* service, coordinating classification operations, asynchronous task scheduling, analytical result management, and dashboard data aggregation through well-defined endpoints. Its implementation follows a layered design structure—*Application*, *Domain*, and *Infrastructure*—according to the principles of Domain-Driven Design (DDD) [8]. This modular separation improves maintainability, scalability, and fault isolation, while fostering a high degree of decoupling among workflow stages. Furthermore, the service provides asynchronous mechanisms for managing execution states—such as scheduling, in-processing, and completion—ensuring consistent task control across distributed components. This architectural design also establishes a robust foundation for extensibility of new analytical capabilities.

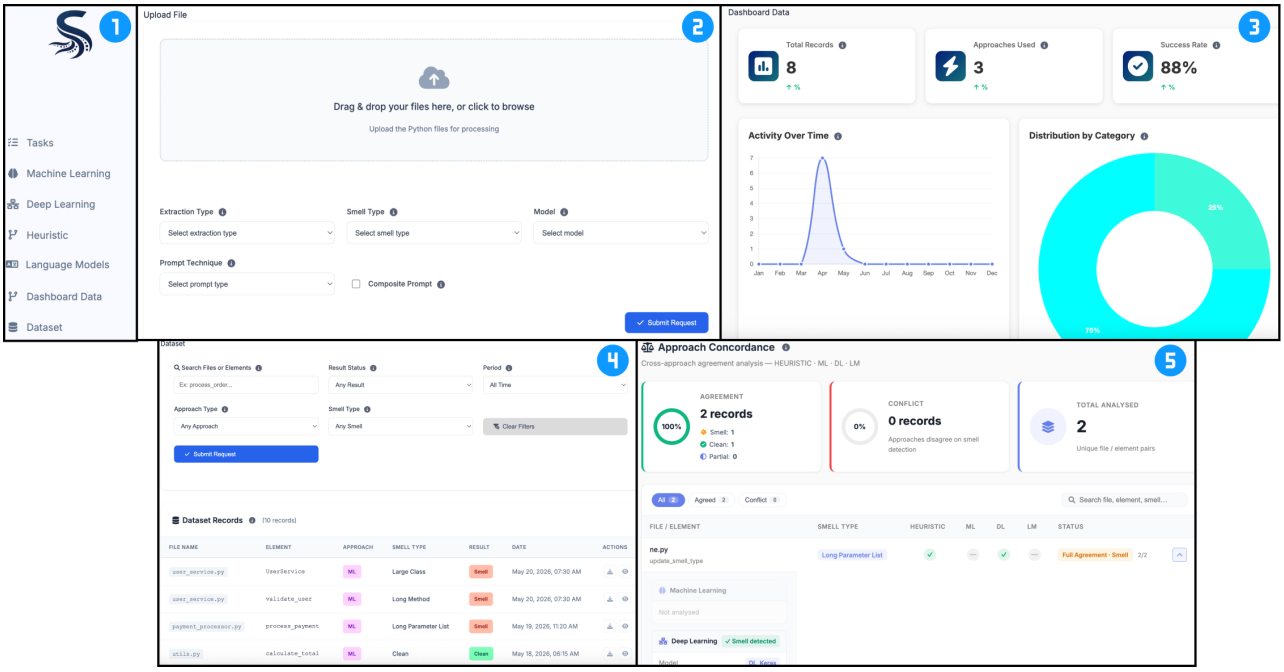

Figure 1: Scylla's screenshot

Table 1: Smells supported by Scylla

Code Smell	Level	Criteria	Metric
Large Class (LC) [3, 5]	Class	$(LOC \geq 200) \vee (NOA + NOM) > 40)$	LOC: Lines of Code; NOA: Number of Attributes; NOM: Number of Methods
Lazy Class (LzC) [32]	Class	$((NOM < 5) \wedge (NOA < 5)) \vee (DIT < 2)$	NOM: Number of Methods; NOA: Number of Attributes; DIT: Depth Inheritance Tree
Data Class (DC) [32]	Class	$(LWMC > 50) \vee (LCOM > 0.8)$	LWMC: Lack of Weighted Method Complexity; LCOM: Lack of Cohesion of Methods
Long Base Class List (LBCL) [5]	Class	$NBC \geq 3$	NBC: Number of Base Classes
Long Method (LoM) [3, 5]	Function	$MLOC \geq 67$	MLOC: Method Lines of Code
Long Parameter List (LPL) [3, 5]	Function	$PAR > 4$	PAR: Number of Parameters
Long Scope Chaining (LSC) [5]	Function	$DOC \geq 3$	DOC: Depth of Closure
Long Message Chain (LMC) [5]	Expression	$LMC \geq 4$	LMC: Length of Message Chain
Long Lambda Function (LLF) [5]	Expression	$NOC \geq 80$	NOC: Number of Characters in One Expression
Complex List Comprehension (CLC) [5]	Expression	$NOL + NOCC \geq 4$	NOL: Number of Loops; NOCC: Number of Control Conditions
Long Element Chain (LEC) [5]	Expression	$LEC \geq 3$	LEC: Length of Element Chain
Long Ternary Conditional Expression (LTCE) [5]	Expression	$NOC \geq 40$	NOC: Number of Characters in One Expression
Magic Number (MN) [3]	Expression	$((val \notin \{0, 1, -1\}) \wedge$ (not ConstantAssignment) $\wedge$ (not ParameterDefault))	NL: Numeric Literal (hard-coded value without semantic definition)
Useless Exception Handling (UEH) [5]	Code Block	$(NEC = 1 \wedge NGEC = 1) \vee (NEC = NEEC)$	NEC: Number of Except Clauses; NGEC: Number of General Exception Clauses; NEEC: Number of Empty Except Clauses

Table 2: LMs supported by Scylla

Type	Model	#P	Quant.	Tags	Size	Context	#T
SLM	Qwen2.5-coder	7.62B	Q4_K_M	dae161e27b0e	4.7 GB	32768	4096
	Codegemma	7.62B	Q4_K_M	dae161e27b0e	5.0 GB	6144	2048
	Mistral	7.25B	Q4_0	f974a74358d6	4.1 GB	32768	4096
	Codellama	6.74B	Q4_0	8fdf8f752f6e	3.8 GB	16384	4096
LLM	Codellama	13B	Q4_0	9f438cb9cd58	7.4 GB	16384	4096
	Deepseek-coder-v2	16B	Q4_0	63fb193b3a9b	8.9 GB	32768	4096
	Qwen2.5-coder	14.8B	Q4_K_M	9ec8897f747e	9.0 GB	32768	4096
	Deepseek-R1	14.8B	Q4_0	c333b7232bdb	9.0 GB	32768	4096

Message Broker: The message-oriented middleware adopted in this architecture is *RabbitMQ* [28], which serves as the central broker for asynchronous communication between system components. Its primary role is to handle the events generated by the classification requests assigned to the LMs. Upon receiving these events from the *code-extractor.API*, the broker queues them for processing by the *lm-integrator* service.

(3) *lm-integrator:* This service was designed to asynchronously process classification tasks and notifications received from the message broker. Operating as an event-driven consumer, it processes code classification tasks through LMs 2, integrated by *Ollama* [23] or accessed through RESTful APIs [11, 18]. These LMs were chosen based on the study proposed by Oliveira et al. [22]. Upon receiving events, the service executes the classification of code fragments and subsequently persists the processed results in the database.

Database: The database adopted in this architecture is *PostgreSQL* [21], serving as the central repository for storing and managing the

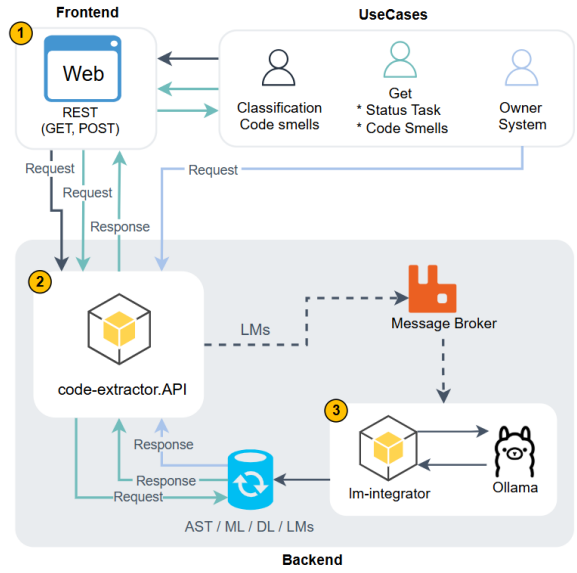

Figure 2: Scylla's architecture

results of code classification tasks. All classification outputs generated for the different analytical approaches—AST-based heuristics, ML, DL, and LMs—are consolidated within this persistence layer, which ensures data consistency, integrity, and reproducibility. Furthermore, this component provides a reliable foundation for data access and retrieval, supporting potential analytical or visualization layers built upon the stored results and facilitating empirical validation of classification accuracy across the different approaches.

4 Study Design

The goal of the defined experiments is to observe and validate Scylla's practical use in supporting the automated classification of code smells. The experiments were structured to reflect realistic usage scenarios, allowing observation of user interactions with Scylla's main functionalities and assessment of their ability to successfully execute classification tasks. The study was organized into three main stages: ① Preparation, ② Execution, ③ Performance Evaluation and ④ Data Analysis.

① *Preparation:* We formulated an experimental framework inspired by a previous study by Castro et al. [4], which guided the structuring of the user tasks into three progressively distinct categories: Filtering Tasks (FT), Basic Tasks (BT), and Assimilation Tasks (AT). This classification was adopted to ensure a systematic and progressive assessment of user interaction with Scylla, ranging from simple operational actions to more complex cognitive reasoning processes. The FT represents straightforward activities designed to confirm the participants' ability to perform fundamental interactions within the tool. The BT evaluates users' understanding of the core functionalities, focusing on their capacity to execute standard operations successfully. Finally, the AT involves more sophisticated reasoning, requiring participants to interpret outputs, correlate information across different analytical approaches, and extract conclusions from the results produced by the system. The complete list of tasks can be found in our complementary material[2].

We conducted a pilot study with two Python developers to evaluate the clarity and feasibility of the task. This stage allowed for adjustments in content, duration, and format, making the instrument more objective and applicable. The implemented improvements reduced the average response time to approx. 60 minutes.

We selected Python developers with different technological backgrounds and varying levels of expertise. The recruitment process was conducted through a preliminary questionnaire aimed at validating each participant's knowledge and confirming their eligibility to take part in the experimentation phase. Moreover, we prepared an informed consent form (ICF) to ensure that all participants were fully aware of the nature and objectives of the study. The document provided a detailed description of the experiment's purpose, the types of data to be collected and how such information would be used to evaluate the proposed tool. Participation in the study was conditional upon the reading and signing of the consent form.

② *Execution:* After providing consent, the execution phase was organized into three stages. First, participants were introduced to the study and informed about the tasks they were required to perform, as well as the procedures to be followed during the experiment. They received a 15-minute overview of Scylla, highlighting its interface and main functionalities. Second, participants engaged in an interview phase where they were asked to perform code classification and analysis tasks while interacting with Scylla and verbalizing their reasoning following the think-aloud protocol. All sessions were recorded via Zoom, capturing both screen activity and audio for later transcription and analysis. Third, upon completing the tasks of the interview, participants answered a post-session questionnaire[3] designed to collect complementary data. It combined 14 open-ended and scaled questions based on the Technology Acceptance Model (TAM) [1], assessing perceived ease of use, usefulness, and overall impressions of the tool, while also including demographic questions to characterize the participants.

③ *Performance Evaluation:* This stage consisted of evaluating the quality and capabilities of Scylla in comparison with existing tools reported in the literature. To achieve this, an investigation of related tools aligned with the scope of this study was conducted, focusing on solutions designed for code smell detection and analysis. Although no tool was identified as capable of integrating heuristic-, ML-, DL-, and LM-based approaches within a unified platform, partially related solutions were found, including the *Dpy* tool proposed by Boloori et al. [3] and the *PyExamine* tool proposed by Shivashankar et al. [29].

To evaluate AST-based heuristic approaches, the software artifacts and benchmark fragments used in the studies of *Dpy* and *PyExamine* were obtained from the datasets and resources publicly released by the respective authors. Subsequently, comparative experiments were conducted using the same code fragments referenced in the original studies to evaluate the code smell classification capabilities of Scylla.

In the context ML-, DL-, LM-based approaches, a dataset composed of code fragments manually labeled by software engineering specialists was employed to evaluate the classification performance of the language models against expert-provided annotations. The manual labeling process was conducted in pairs to ensure greater

[2] lm4smells-core/docs/tasks

[3] lm4smells-core/docs/questionnaire

annotation reliability. In addition, Cohen's Kappa coefficient was employed to measure the level of inter-rater agreement during the labeling process, as adopted in previous studies [33]. In cases of disagreement between the specialists, a third validation stage was conducted to resolve conflicts, increasing the rigor and consistency of the generated ground truth.

④ *Data Analysis:* The final stage of the study focused on analyzing the collected results. This phase examined participants' responses to assess Scylla's performance and extract empirical evidence supporting its quality. We computed metrics such as, task completion rate, accuracy, and time spent per section. Moreover, we analyzed participants' perceptions about Scylla's use.

We defined three research questions (RQs), as follows:

RQ_1 How effectively can Scylla support users in performing tasks of varying complexity within realistic code analysis and classification scenarios? This question examines Scylla usability as experienced by participants during the execution of experimental tasks designed to replicate realistic code analysis workflows. Participants interacted with Scylla through filtering, basic, and assimilation tasks, each varying in cognitive demand and required interface interaction.

RQ_2. How do users perceive the usefulness and practical applicability of Scylla? This question focuses on capturing participants' evaluation of Scylla's relevance, clarity, and alignment with their analytical needs. To address it, we analyzed responses from a structured post-session questionnaire.

RQ_3. How does the proposed tool compare with existing tools? This question investigates how Scylla compares with existing tools in the literature regarding classification performance for Python code smell detection.

5 Results and Discussions

A total of nine participants contributed to the study. Most participants are specialized in Software Engineering (88.9%), followed by Data Science (33.3%), and AI (11.1%). Regarding professional experience in software development, 44.4% of participants reported 1–3 years of experience, 33.3% reported 4–6 years, while the remaining indicated more than 7 years (both 11.1%). Concerning familiarity with code smell classification tools, 66,7% indicated high to moderate knowledge. When asked about the main advantages of using tools for code smell classification, participants identified several benefits. #P1, #P2, #P5, #P6, and #P9 mentioned *"tools are valuable mechanisms for enhancing software quality and supporting developers in producing cleaner, more maintainable code"*. #P1 and #P2 emphasized their *"usefulness in guiding less-experienced professionals toward recognizing and correcting design deficiencies"*. P3, P4 and P5 mentioned their *"role as auditors for expediting refactoring processes and mitigating the spread of structural issues in large-scale systems"*. Additionally, #P5, #P7, and #P8 noted that *"these tools enable analytical operations that would be hard to perform manually, particularly those involving complex metrics"*. Collectively, these perceptions indicate that participants are aware that automated code smell classification tools contribute to software quality.

5.1 Scylla's Effectivity (RQ_1)

To address RQ_1, we examined participants' performance during the experimental tasks The tasks were divided into three progressive categories—FT, BT, and AT—to assess the effectiveness of Scylla across increasing levels of cognitive and operational complexity. Overall, the results demonstrate a high level of task completion accuracy, reflecting Scylla's clarity, consistency, and reliability throughout different interaction stages. The detailed table results are available in our supplementary material[4].

Filtering tasks (FT1–FT5) showed the highest completion rates across all participants, indicating that participants correctly understood the essential functioning of the tool. Participants were able to navigate the interface intuitively, with minimal cognitive load. The completion times ranged from approximately 2min to 10min, reinforcing the tool's efficiency in handling exploratory tasks. Although overall performance was high, two isolated errors were identified. In FT1, participants #P4 and #P7 had difficulty in recognizing all classification approaches supported by the tool. These misunderstandings indicate interpretive nomenclature issues in the tool, since both participants mixed the names of the approaches with the names of the supported extraction types. Participant #P4 mentioned: *"Well, I think I should look here under Language Models. And inside Language Models, I need to check the supported extraction and smell type…"*, a comment that illustrates that the participant knew the tool supported the Language Model approach, but she was uncertain about what, in fact, meant an approach.

Basic tasks (*BT1–BT6*) also achieved strong results, with participants successfully executing classifications using all analytical approaches—AST, ML, and LMs—and exporting results without major difficulty. Participants #P1 and #P4 experienced difficulty understanding precisely what actions needed to be performed for the task *BT1*, resulting in responses that were inconsistent with the task's objectives. In *BT5*, participants #P3-#P5 and #P7 struggled to correctly identify the exported files, compromising the accuracy of the expected answer. Completion times ranged between 10min and 15min, except for participant #P9, who took nearly 21min, reinforcing that task execution requires attention but remains manageable. Note that in these tasks, there is also the inherent processing time associated with model inference and task scheduling, and despite that, these outcomes indicate that Scylla enables users to perform the core code-classification operations efficiently.

Assimilation tasks (*AT1–AT3*) represented the most complex tasks, requiring comparative reasoning between approaches and interpretation of model explainability outputs. Although participants completed most tasks successfully, most of them committed at least one error. These suggest opportunities to enhance the way comparative analyses are presented to users, particularly regarding the visual organization and clarity of information derived from multiple classification approaches. For instance, the addition of an analytical dashboard could significantly strengthen support for comparative reasoning tasks, enabling users to explore differences across approaches more intuitively and systematically.

5.2 Scylla's Usefulness and Applicability (RQ_2)

To address RQ_2, we analyzed participants' perceptions of Scylla's usefulness and practical applicability through a post-session questionnaire. The aggregated scores are presented in Figure 3. Overall,

[4] lm4smells-core/docs/results

the responses revealed a positive evaluation, particularly regarding the clarity of its interface and the intuitiveness of its workflows.

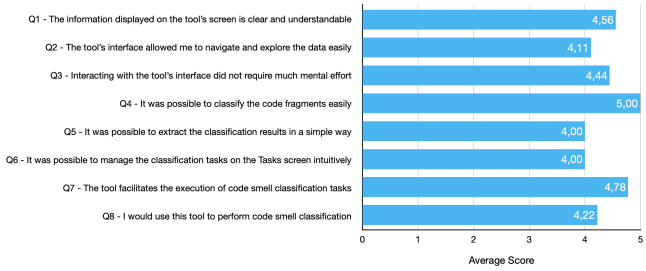

Figure 3: Average participant scores for each question

Questions *Q1*, *Q2*, and *Q3*, which assessed clarity, ease of navigation, and cognitive effort, achieved strong ratings (4.56, 4.11, and 4.44, respectively), indicating that participants found the interface clear, consistent, and cognitively accessible. Similarly, *Q4* and *Q7* about the classification process received the highest mean score (5.00 and 4.78), suggesting that the classification process was straightforward and well-integrated into `Scylla`'s design.

Regarding operational aspects, *Q5* and *Q6*—which evaluated the simplicity of extracting results and managing classification tasks—both obtained average scores of 4.00. In *Q5*, participants reported difficulties related to the extraction and interpretation of results. For instance, *#P5* noted that *"the extraction is not easy due to the lack of information on the download screen; comparing two approaches also requires manual work in external tools."* *#P7* reinforced these concerns, commenting that *"evaluating the result requires data analysis, which can be complex when done quickly through CSV"*. For *Q6*, participants expressed issues related to task identification and management. *#P4* reported that *"it was hard to identify which file was which, since the file names did not clearly differentiate the smell type"*. Participants also mentioned difficulties handling multiple files, as *#P9* explained: *"managing results from different approaches is hard when relying only on task/file names"*. Finally, *Q8*, which measured the intention to use `Scylla` in real scenarios, achieved a mean score of 4.22. Collectively, these findings evidence the strong acceptance and perceived utility of `Scylla` among participants, demonstrating high potential for its adoption.

5.3 `Scylla` Compared to Existing Tools (RQ_3)

To address RQ_3, we compared `Scylla` with two existing Python code smell detection tools, DPy and PyExamine. We used as ground truth the `PyHub-SMELL` dataset [7] with a subset of two smell types supported by all three tools: LoM and LPL. In this subset, `Scylla` obtained a micro precision of 0.6996, micro recall of 0.2611, and micro F1 of 0.3803. PyExamine obtained higher precision than DPy, with micro precision of 0.9605, micro recall of 0.1524, and micro F1 of 0.2631, compared with 0.8871, 0.1148, and 0.2033 for DPy, respectively. While PyExamine and DPy showed high precision but low recall, `Scylla` achieved the highest recall and F1-score, indicating a better balance between identifying positive instances and limiting false positives.

Using the same benchmark annotated dataset, we also analyzed the classification approaches integrated into `Scylla`, namely heuristic rules, ML, DL, and LMs to examine their execution and comparison within a unified experimental environment. The heuristic approach showed the most conservative behavior, achieving a precision of 0.6996 and recall of 0.2611, with 170 true positives and 481 false negatives. Although it missed many positive instances, it provides a deterministic baseline. Among the learned approaches, ML achieved the highest precision of 0.8098, with a recall of 0.6958, while DL obtained the highest recall of 0.7465, with a precision of 0.7703. Thus, ML was more selective, while DL identified a larger proportion of positive instances. The LM-based approach achieved a precision of 0.7577 and recall of 0.5839, with 9,545 (out of 16,347 instances) true positives. Although its recall was lower than that of ML and DL, it classified substantially more instances, supporting its applicability to broader scenarios.

In general, the approaches exhibited complementary behavior, with ML and DL offering more balanced performance, and LMs enabling larger-scale classification. Comparison with DPy and PyExamine also revealed conservative behavior among AST-based tools, strengthening the value of an infrastructure that supports multiple classification strategies rather than a single detection paradigm.

All results are summarized in our complementary material [6], including true positives, false positives, false negatives, true negatives, precision, and recall.

6 Conclusion and Future Work

We presented `Scylla`, a tool that unifies four approaches to code smell classification—AST-based heuristics, ML, DL, and LMs—within a single accessible environment. By allowing the submission, classification, and export of results through a structured workflow, `Scylla` supports reproducible and multifaceted analyzes of software quality. Our experiments showed that the participants successfully completed tasks of varying complexity, demonstrating that `Scylla` provides a clear and intuitive interaction flow.

Future work will focus on four main directions. First, we plan to integrate `Scylla` with GitHub, allowing users to analyze repositories directly from the platform. Second, we also plan to investigate explainable AI techniques that provide richer justifications for classifications and actionable refactoring recommendations. Third, we aim to extend `Scylla` to additional code smells and programming languages (C#, Go, JavaScript, and Java), broadening its applicability across diverse development ecosystems. Ultimately, we envision `Scylla` as an extensible research platform that fosters reproducible experimentation and advances intelligent software quality analysis.

Artifact Availability

The artifacts are available at https://zenodo.org/records/21461946.

Acknowledgments

This research was partially funded by the Brazilian funding agencies CAPES/COFECUB (Grant 88881.879016/2023-01 and Ma1036/24), CNPq (Grant 406089/2025-6), FAPESP (Grant 2023/00811-0), and FAPEMIG (Grant APQ-01488-24). We also acknowledge the Brazilian company Stone[5] for their financial support.

[5]https://www.stone.com.br/

References

[1] Emran Aljarrah, Hamzah Elrehail, and Bashar Aababneh. 2016. E-voting in Jordan: Assessing readiness and developing a system. *Computers in Human Behavior* 63 (2016), 860–867. doi:10.1016/j.chb.2016.05.076

[2] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology* 108 (2019), 115–138. doi:10.1016/j.infsof.2018.12.009

[3] Aryan Boloori and Tushar Sharma. 2025. DPy: Code Smells Detection Tool for Python. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. 826–830. doi:10.1109/MSR66628.2025.00119

[4] Diego Castro and Marcelo Schots. 2018. Analysis of test log information through interactive visualizations. In *Proceedings of the 26th Conference on Program Comprehension*. Association for Computing Machinery, 156–166. doi:10.1145/3196321.3196345

[5] Zhifei Chen, Lin Chen, Wanwangying Ma, and Baowen Xu. 2016. Detecting Code Smells in Python Programs. In *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*. 18–23. doi:10.1109/SATE.2016.10

[6] Igor Soares de Oliveira, Alexandre Andrade, Rayssa Athayde, Eduardo Nadai, and Juliana Alves Pereira. 2026. *Scylla: A Unified Tool for Code Smell Classification in Python*. doi:10.5281/zenodo.21461946

[7] Igor Soares de Oliveira, Alexandre Andrade, Saymon Souza, Eduardo Figueiredo, and Juliana Alves Pereira. 2026. A Multi-Approach Evaluation of Python Code Smell Detection Using Heuristics, Learning Models, and Language Models. In *Anais do Simpósio Brasileiro de Engenharia de Software* (São Paulo/SP). SBC, São Paulo, SP, Brasil, 1–12.

[8] Eric Evans. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

[9] David Flanagan. 2020. *JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language* (7 ed.). O'Reilly Media.

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. 2012. *Refactoring: Improving the Design of Existing Code*. Pearson Education.

[11] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing RESTful APIs: A Survey. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 27 (Nov. 2023), 41 pages. doi:10.1145/3617175

[12] Peter Hamfelt, Ricardo Britto, Lincoln Rocha, and Camilo Almendra. 2025. Automatic Identification of Machine Learning-Specific Code Smells. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 337–347. doi:10.5753/sbes.2025.9949

[13] Anh Ho, Anh M.T. Bui, Phuong T. Nguyen, Amleto Di Salle, and Bach Le. 2025. EnseSmells : Deep ensemble and programming language models for automated code smells detection. *Journal of Systems and Software* 224 (2025), 112375. doi:10.1016/j.jss.2025.112375

[14] Michele Lanza and Radu Marinescu. 2006. Object-Oriented Metrics in Practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. (2006). doi:10.1007/3-540-39538-5

[15] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. 2021. Deep Learning Based Code Smell Detection. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1811–1837. doi:10.1109/TSE.2019.2936376

[16] M. Mantyla, J. Vanhanen, and C. Lassenius. 2003. A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* 381–384. doi:10.1109/ICSM.2003.1235447

[17] R.C. Martin. 2009. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.

[18] M. Masse. 2011. *REST API Design Rulebook*. O'Reilly Media.

[19] Djamel Mesbah, Nour El Madhoun, Khaldoun Al Agha, and Hani Chalouati. 2025. Leveraging Prompt-Based Large Language Models for Code Smell Detection: A Comparative Study on the MLCQ Dataset. (2025), 444–454. doi:10.1007/978-3-031-86149-9_42

[20] Eric Meyer and Estelle Weyl. 2023. *CSS: The Definitive Guide: Web Layout and Presentation* (5 ed.). O'Reilly Media.

[21] Regina O. Obe and Leo S. Hsu. 2017. *PostgreSQL: Up and Running: A Practical Guide to the Advanced Open Source Database* (3 ed.). O'Reilly Media.

[22] Igor Oliveira, Joanne Carneiro, Jessica Ribas, and Juliana Pereira. 2025. Code Smell Classification in Python: Are Small Language Models Up to the Task?. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software* (Recife/PE). SBC, 699–705. doi:10.5753/sbes.2025.11046

[23] Ollama. 2025. Ollama. https://ollama.com/ Accessed on: July 2, 2025.

[24] PyCQA. 2025. *Flake8: The modular source code checker for Python*. https://github.com/PyCQA/flake8 Accessed: 2025-11-10.

[25] PyCQA. 2025. *Pyflakes: A simple program which checks Python source files for errors*. https://github.com/PyCQA/pyflakes Accessed: 2025-11-10.

[26] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[27] Rajwant Singh Rao, Seema Dewangan, and Alok Mishra. 2025. An Empirical Evaluation of Ensemble Models for Python Code Smell Detection. *Applied Sciences* 15, 13 (2025). doi:10.3390/app15137472

[28] Gavin M. Roy. 2017. *RabbitMQ in Depth*. Manning Publications.

[29] Karthik Shivashankar and Antonio Martini. 2025. PyExamine: A Comprehensive, Un-Opinionated Smell Detection Tool for Python. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 763–774. doi:10.1109/MSR66628.2025.00114

[30] Raúl Tabarés. 2021. HTML5 and the evolution of HTML; tracing the origins of digital platforms. *Technology in Society* 65 (2021), 101529. doi:10.1016/j.techsoc.2021.101529

[31] The Pylint Team. 2024. Pylint — Static Code Analysis for Python. https://pylint.pycqa.org/.

[32] Fatma Neda Topuz and David A. Umphress. 2023. Do Bad Smells Lead to Defects?. In *2023 13th International Conference on Software Technology and Engineering (ICSTE)*. 75–81. doi:10.1109/ICSTE61649.2023.00020

[33] Susana M. Vieira, Uzay Kaymak, and João M. C. Sousa. 2010. Cohen's kappa coefficient as a performance measure for feature selection. In *International Conference on Fuzzy Systems*. 1–8. doi:10.1109/FUZZY.2010.5584447

[34] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2023. AUTOMATIC CHAIN OF THOUGHT PROMPTING IN LARGE LANGUAGE MODELS. *11th International Conference on Learning Representations, ICLR 2023* (2023). https://www.scopus.com/inward/record.uri?eid=2-s2.0-85161923913&partnerID=40&md5=ccae1b0019daf31478edb17dfbab078c